Answering Conjunctive Queries and FO+MOD Queries under Updates

Dissertation
zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät der Humboldt-Universität zu Berlin

von

M.Sc. Jens Keppeler

Präsidentin der Humboldt-Universität zu Berlin

Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:

Prof. Dr. Elmar Kulke

Gutachter/innen :	1. Prof. Dr. Nicole Schweikardt, Humboldt-Universität zu Berlin 2. Prof. Dr. Stefan Kratsch, Humboldt-Universität zu Berlin 3. Prof. Dr. Wim Martens, Universität Bayreuth
Tag der mündlichen Prüfung:	25. Mai 2020

Abstract

This thesis investigates the query evaluation problem for fixed queries over fully dynamic databases, where tuples can be inserted or deleted. The task is to design a dynamic algorithm that immediately reports the new result of a fixed query after every database update.

In particular, the goal is to construct a data structure that allows to support the following scenario: after every database update, the data structure can be updated in constant time such that afterwards we are able to test within constant time for a given tuple whether or not it belongs to the query result (the testing routine), to output the number of tuples in the query result (the counting routine), to enumerate all tuples in the new query result (the enumeration routine), and to enumerate the difference between the old and the new query result with constant delay (the difference routine). The preprocessing time needed to build the data structure is linear in the size of the database.

In the first part of this thesis, conjunctive queries on arbitrary relational databases are considered. The notion of *q-hierarchical* conjunctive queries is introduced and it is shown that the result of each such query on a dynamic database can be maintained efficiently in the sense described above. Moreover, this notion is extended to aggregate queries for which we can also maintain the query result under updates. Furthermore, it is shown that the preparation of learning a polynomial regression function can be done in constant time if the training data are taken (and maintained under updates) from the query result of a q-hierarchical query. For the testing problem, the notion of t-hierarchical conjunctive queries, a more expressive query language, is considered and it is shown that such queries can be maintained efficiently and allow to test whether or not a given tuple belongs to the result set. It turns out that if one allows logarithmic update time in the size of the database (and $n \log(n)$ preprocessing time as well), then the data structure for q-hierarchical queries additionally supports the following routine: upon input of a natural number j, output the jth tuple that will be enumerated by the enumeration routine. Furthermore, the notion of q-hierarchical and t-hierarchical conjunctive queries is , ifed to unions of conjunctive queries (UCQs) and it is shown that there is a data structure that can be maintained under updates, supports the enumeration routine and the testing routine, and a subset of q-hierarchical UCQs is considered for which one can output the jth solution of a enumeration.

In the second part of this thesis, queries in first-order logic (FO) and its extension with modulo-counting quantifiers (FO+MOD) are considered, and it is shown that they can be efficiently evaluated under updates, provided that the dynamic database does not exceed a certain degree bound, and the counting, testing, enumeration and difference routines is supported.

Zusammenfassung

In dieser Arbeit wird das dynamische Auswertungsproblem für feste Anfragen über dynamischen Datenbanken betrachtet, bei denen Tupel hinzugefügt oder gelöscht werden können. Die Aufgabe besteht darin, einen dynamischen Algorithmus zu konstruieren, welcher unmittelbar nachdem die Datenbank aktualisiert wurde, eine Datenstruktur aktualisiert, die das Resultat ändert.

Darüber hinaus soll die Datenstruktur in konstanter Zeit aktualisiert werden und folgende Routinen unterstützen. Nach jeder Datenbankaktualisierung kann die Datenstruktur in konstanter Zeit angepasst werden, so dass anschließend in konstanter Zeit getestet werden kann, ob ein Tupel zur Ausgabemenge gehört (die Test-Routine), die Anzahl der Tupel in der Ausgabemenge in konstanter Zeit ausgeben werden kann (die Anzahl-Routine), die Tupel aus der Ausgabemenge mit konstanter Taktung aufgezählt werden kann (die Aufzähl-Routine), und der Unterschied zwischen der neuen und der alten Ausgabemenge mit konstanter Taktung aufgezählt werden kann (die Unterschied-Routine). Die Vorverarbeitungszeit um die Datenstruktur aufzubauen benötigt lineare Zeit in der Größe der Datenbank.

Im ersten Teil dieser Arbeit werden konjunktive Anfragen auf beliebigen relationalen Datenbanken betrachtet. Die Idee der *q-hierarchischen* Anfragen wird eingeführt und es wird gezeigt, dass das Resultat für jede q-hierarchische Anfrage auf dynamischen Datenbanken effizient ausgewertet werden können in dem oben beschriebenen Szenario. Darüber hinaus wird dieses Szenario auf q-hierarchische Anfragen mit Aggregaten erweitert und es wird gezeigt, dass das gleiche Szenario ebenfalls effizient gelöst werden kann. Außerdem wird gezeigt, dass das Lernen von polynomiellen Regressionsfunktionen in konstanter Zeit vorbereitet werden kann, falls die Trainingsdaten aus dem Anfrageergebnis einer q-hierarchischen Anfragen kommen und insbesondere in konstanter Zeit aktualisiert werden können, wenn sich die Datenbank ändert. Für die Test-Routine betrachten wir die t-hierarchischen Anfragen, die ausdrucksstärker als q-hierarchische Anfragen sind, für die wir die Test-Routine auf dynamische Datenbanken auswerten können. Es stellt sich heraus, dass wenn man logarithmische Zeit in den Aktualisierungen für q-hierarchische Anfragen zulässt (und insbesondere $n \log(n)$ Vorverarbeitungszeit erhält), man auch folgende Routine realisieren kann: Bei Eingabe einer natürlichen Zahl j, gib das $j$te Tupel von dem Aufzählalgorithmus aus. Außerdem, werden die Ideen von q-hierarchischen und t-hierarchischen konjunktive Anfragen auf Vereinigungen konjunktive Anfragen (UCQs) erweitert und es wird gezeigt, dass es eine Datenstruktur für diese Anfragen auf dynamische Datenbanken gibt, die mit konstanter Aktualisierungszeit, die Aufzähl-Routine und die Test-Routine unterstützt. Darüber hinaus wird das Problem für die Ausgabe des $j$ten Tupel auf eine Teilmenge für q-hierarchische Anfragen betrachtet.

Im zweiten Teil der Arbeit werden Anfragen, die Formeln der Logik erster Stufe (FO)

und deren Erweiterung mit Modulo-Zähl Quantoren (FO+MOD) sind, betrachtet, und es wird gezeigt, dass diese effizient unter Aktualisierungen ausgewertet können, wobei die dynamische Datenbank die Gradschranke nicht überschreitet, und bei der Auswertung die Zähl-, Test-, Aufzähl- und die Unterschied-Routine unterstützt werden.

Contents

1. Introduction

In this thesis we consider the algorithmic task of evaluating a query on fully dynamic databases, where tuples can be inserted or deleted. More generally, the aim is to design a data structure representing the query and the database, which can be updated every time a tuple will be inserted into or removed from the database and it solves at least one of following problems efficiently:

- answer Boolean queries (*answering problem*),
- enumerate the tuples in the result set without repetition with constant delay (*enumeration problem*),
- on input of a tuple, test if the tuple belongs to the result set (*testing problem*),
- enumerate the tuples that were added to / removed from the result set after the last update step (*diff-and-report problem*),
- enumerate the tuples that are added in / removed from the result set after some update steps (*difference problem*),
- output the number of tuples in the result set (*counting problem*),
- prepare the learning of the parameters for a polynomial regression function where the training set is taken from the query result (*learning problem*) and
- given a natural number j, output the jth tuple in the enumeration, if it exists (j*th tuple problem*).

We consider finite relational databases over a possibly infinite domain.

In the beginning, we obtain an initial database D_0 and a query Q. In the *preprocessing phase* we construct the data structure that represents the database D_0 and Q. Whenever we change the database, i.e., a tuple is being inserted or deleted, we modify the data structure in an *update step* such that it represents the updated database. The *update time* is the time needed to process an update step. In order to be efficient, our aim is to show that the update steps take constant or logarithmic time in the size of the database and in particular, the update time is way smaller than the time needed to recompute the entire query result.

This thesis is divided into two parts.

The first part's aim is to give algorithms for a subclass of conjunctive queries (CQ for short), unions of conjunctive queries (UCQ for short), and conjunctive queries

with aggregates, all of which can be efficiently maintained under updates. It turns out that for the answering problem, the enumeration problem, the difference problem and the counting problem, the class of *q-hierarchical* queries are suitable for solving these problems under updates. The notion of q-hierarchical queries are strongly related to the *hierarchical* property that was introduced by Dalvi and Suciu in [34] and already played a central role for efficient query evaluation in various contexts (see Chapter 3 for a definition). The following is shown: after a linear time preprocessing phase (here we consider data complexity, i.e., linear time means linear in the size of the database) the q-hierarchical query can be maintained under updates and can solve the answering problem, the enumerating problem, the counting problem and the difference problem (see Theorem 3.3). This means that after every update the data structure can still solve these problems. It turns out that the testing problem can be done efficiently under updates for a superclass of q-hierarchical queries after a linear time preprocessing. We call these conjunctive queries t-hierarchical (see Definition 3.4 for a definition and Theorem 3.6 for the result).

Furthermore, we also consider extensions of conjunctive queries, queries with aggregates and unions of conjunctive queries. We define a class of conjunctive queries with aggregates, such that we can maintain queries (by solving the answering, enumerating, testing and the counting problem) under updates after a linear time preprocessing phase (see Theorem 3.7). We show that every q-hierarchical query with aggregates can be reduced to q-hierarchical conjunctive queries without aggregates. In particular, it follows that if a problem is tractable under updates for q-hierarchical conjunctive queries, then it is also tractable for q-hierarchical CQs with aggregates if we can maintain the corresponding aggregate functions in the query under updates in sufficient time. That means, for example, we only obtain constant update time, if we can update the values of the aggregation function for the result set in constant time.

As mentioned before, we consider the problem of leaning a polynomial regression function, a problem which comes from the topic of *supervised machine learning*. The main idea of this topic is to find relations or patterns between data of a given training set. After using an algorithm that learns a model ("learn" means in this context to compute the model) we can use the model to obtain approximate solutions of other inputs. In this thesis, we consider the model of a polynomial regression, which is a polynomial function that minimizes the error between the solutions in the training set and the value of the function. Such a learning algorithm has two passes, where in the first pass precomputations are done for the second pass. In the second pass, numerical methods were used to approximate the parameters (for more details see Chapter 7). In this thesis we show how to efficiently maintain the computations that are done in the first pass under updates where the training set is the query result (see Theorem 3.8).

For a nice survey to machine learning see [54, 43].

For the task of solving the jth problem, we consider logarithmic update time rather than constant time. We show that after an $n \log(n)$ time preprocessing phase (here n is the size of the database), we can output for a given natural number j the jth tuple in the result of a q-hierarchical CQ with or without aggregates on a database D under logarithmic updates (if the aggregates can be computed in logarithmic time or faster)

(see Theorem 3.9).

For unions of conjunctive queries we obtain results for the testing problem for a union of t-hierarchical conjunctive queries (this is a UCQ where every CQ in the query is t-hierarchical) and the enumeration problem for q-hierarchical UCQ (a UCQ where every CQ in the query is q-hierarchical) under updates. Furthermore, we consider the problem for reporting the jth tuple for UCQ. We identify a class of UCQs for which we can maintain this problem efficiently under updates. We call these queries strongly exhaustively q-hierarchial UCQs (see Theorem 3.12).

In the second part of this thesis, we consider queries in first-order logic (FO) and its extension with modulo-counting quantifiers (FO+MOD) and show that they can be efficiently evaluated under updates, provided that the dynamic database does not exceed a certain degree bound. We obtain in the second part the following result: let Q be a k-ary FO+MOD-query and d a degree bound on the database. Q and d are assumed to be fixed, i.e., in contrast to the database they do not change. With $\|Q\|$ ($\|D\|$) we denote the size of a query Q (database D, resp.). On input of an initial database D_0, we can construct in a linear time $f(\|Q\|, d)\|D\|$ preprocessing phase a data structure that can be updated in time $f(\|Q\|, d)$ and allows to

- immediately answer Q on D if Q is a Boolean query,
- test for a given tuple whether it belongs to the result set in time $O(k^2)$,
- immediately output the number of tuples in the result set $Q(D)$,
- enumerate the tuples in the result set $Q(D)$ with delay $O(k^2)$ and
- enumerate the tuples in $Q(D^-) \setminus Q(D^+)$ and $Q(D^+) \setminus Q(D^-)$ with delay $O(k^2)$, where D^- and D^+ denote the database before and after performing the update operation, respectively.

The function $f(Q, d)$ stands for a function of the form

$$f(Q, d) \quad = \quad 2^{d^{2^{O(\|Q\|)}}}.$$

In contrast to the difference enumeration in the thesis' first part, we enumerate here the difference immediately after exactly one update step.

The dynamic query evaluation algorithm crucially relies on the locality of FO+MOD and, in particular, an effective Hanf normal form for FO+MOD on databases of bounded degree recently obtained by Heimberg, Kuske and Schweikardt [55]. The organization of the second part is given in Chapter 10.

1.1. The author's contribution of the published articles

The articles [15],[17],[19], [16],[18],[21] and [20] were published with co-authors, namely Christoph Berkholz and Nicole Schweikardt. The author contacted his co-authors to reaffirm that there are no disputes over the ownership of the material presented in this thesis.

[15] is a preprint of [16]. The author's main contribution in [16] lies on the upper bounds that are specified in Chapter 1,2,3,4,6 and 7 in [16]. Moreover, there are extensions of these results that are presented in this thesis.

[19] is a preprint of [21]. The author's essential contribution are the upper bound for testing t-hierarchical conjunctive queries and t-hierarchical unions of conjunctive queries and the upper bounds for enumerating q-hierarchical unions of conjunctive queries. These results can be found in Chapter 1,2,3 and 4 in [21]. Moreover, the queries in [16] did not consider constants whereas in [21] the result was lift up to q-hierarchical queries with constants. In this thesis, q-hierarchical queries were introduced with constants and the results are shown with considering constants.

In Chapter 3 the main theorems of the first part are given together with a description which parts of the main theorem are results from [16] and [21] and which parts are new results.

[17] is a preprint of [18] and published in a journal [20]. This work was made in a close cooperation with the co-authors and the contribution of the authors is to be recognized equally. Part II is closely based on the journal version [20].

1.2. Related work

There is a lot of work known that studies the complexity of query evaluation in the static setting. Since there is a huge amount of such work, surprisingly little is know about evaluating queries under updates. The task of answering queries against a dynamic database has been studied under the name of *incremental view maintenance* (see e.g. [53, 30, 66, 68, 81]).

There is a framework, introduced by Patnaik and Immerman [83], called the *dynamic descriptive complexity* framework, which focuses on the expressive power of first-order logic on dynamic databases (see [88] for a survey). Updating this approach may take polynomial time (even in the the case of conjunctive queries [96]). This setting is too expensive in the area for dynamic algorithms.

Some work in computation complexity of query evaluation under updates has been done by Björklund, Gelade and Martens [22] for XPath evaluation and for MSO queries on trees by Balmin, Papakonstantinou and Vianu [11], by Losemann and Martens [74], by Niewerth and Segoufin [80] and by Amarilli et al. [4].

In [59], the enumeration and testing problem under updates has been studied for q-hierarchical and (more general) acyclic CQs in a setting that is very similar to the thesis' first part setting. The *Dynamic Constant-delay Linear Representations* (DCLR) of [59] are data structures that use at most linear update time and solve the enumeration problem and the testing problem with constant delay and constant

testing time.

In case of the static setting, a lot research has been done. Below, there is an overview of known results.

Complexity of Boolean Queries. The complexity of answering Boolean conjunctive queries on a static database is fairly well understood. For every fixed database schema σ, extending a result of [51], Grohe [47] gave a tight characterisation of the *tractable* CQs under the complexity theoretic assumption $\mathsf{FPT} \neq \mathrm{W}[1]$: If we are given a Boolean CQ Q of size $\|Q\|$ and a σ-database D of size $\|D\|$, then Q can be answered against D in time $f(\|Q\|) \cdot \|D\|^{O(1)}$ for some computable function f if and only if the homomorphic core of Q has bounded treewidth. Marx [76] extended this classification to the case where the schema is part of the input.

Counting Complexity. For computing the number of output tuples of a given join query (i.e., a quantifier-free CQ) over a fixed schema σ, a characterisation was proven by Dalmau and Jonsson [33]: assuming $\mathsf{FPT} \neq \#\mathrm{W}[1]$, the output size $|Q(D)|$ of a join query Q evaluated on a σ-database D of size $\|D\|$ can be computed in time $f(\|Q\|) \cdot \|D\|^{O(1)}$ if and only if Q has bounded treewidth. The result has recently been extended to all conjunctive queries over a fixed schema by Chen and Mengel [28]. Structural properties that make the counting problem for CQs tractable in the case where the schema is part of the input have been identified in [37, 46]. The counting problem, i.e., output the number in the result set on input of a database and a query, has been studied for conjunctive queries [28] and for existential positive formulas [29]. In [33] Dalmau and Jonsson showed that the problem of counting the number of homomorphisms in a class $\mathcal{C}$ of structures to a given arbitrary structure is solvable if and only if all structures in $\mathcal{C}$ have bounded treewidth.

Join Evaluation. When the entire result of a non-Boolean query has to be computed, the evaluation problem cannot be modelled as a decision or counting problem and one has to come up with different measures to characterise the hardness of query evaluation. One approach that has been fruitfully applied to join evaluation is to study the worst-case output size as a measure of the hardness of a query. Atserias, Grohe and Marx [6] identified the fractional edge cover number of the join query as a crucial measure for lower bounding its worst-case output size. This bound was also shown to be optimal and is matched by so called "worst-case optimal" join evaluation algorithms, see [79, 95, 78, 63].

Query Enumeration. Another way of studying non-Boolean queries that is independent of the actual or worst-case output size is *query enumeration*. A query enumeration algorithm evaluates a non-Boolean query by reporting, one by one without repetition, the tuples in the query result. The crucial measure to characterise queries that are tractable w.r.t. enumeration is the delay between two output tuples. In the context of constraint satisfaction, the combined complexity, where the query as well as the database are given as input, has been considered. As the size of the query result might be exponential in the input size in this setting, queries that can be enumerated with *polynomial delay* and polynomial preprocessing are regarded as "tractable". Classes of conjunctive queries that can be enumerated with polynomial delay have

been identified in [24, 45]. However, a complete characterisation of conjunctive queries that are tractable in this sense is not in sight.

More relevant to the database setting, where one evaluates a small query against a large database, is the notion of *constant delay enumeration* introduced by Durand and Grandjean in [36]. The preprocessing time is supposed to be much smaller than the time needed to evaluate the query (usually, linear in the size of the database) and the delay between two output tuples may depend on the query, but not on the database. A lot of research has been devoted to this subject, where one usually tries to understand which structural restrictions on the query or on the database allow constant delay enumeration. For an introduction to this topic and an overview of the state-of-the-art the author refers the reader to the surveys [92, 91, 90].

Bagan, Durand and Grandjean [8] showed that acyclic conjunctive queries that are *free-connex* can be enumerated with constant delay after a linear time preprocessing phase (cf. [23] for a simplified proof of their result). They also showed that for self-join free acyclic conjunctive queries the free-connex property is essential by proving the following lower bound. Assume that multiplying two $n \times n$ matrices cannot be done in time $O(n^2)$. Then the result of a self-join free acyclic conjunctive query that is not free-connex cannot be enumerated with constant delay after a linear time preprocessing phase.

It turns out that the notion of q-hierarchical conjunctive queries is a proper subclass of the free-connex conjunctive queries. Thus, there are queries that can be efficiently enumerated in the static setting but are hard to maintain under database updates.

Carmeli and Kröll generalized the notion of free-connex to UCQs [26] and they showed that they are tractable (in the sense that the result set can be enumerate with constant delay after a linear time preprocessing phase) and moreover, they showed that there are UCQs that are not free-connex but tractable. The problem of finding the subclass of UCQ, that are tractable in the static setting still remains open.

First-order query evaluation on static databases. In the thesis' second part, the query language $\mathsf{FO}{+}\mathsf{MOD}$, the extension of first-order logic FO with modulo-counting quantifiers of the form $\exists^{i \bmod m} x\, \psi$, expressing that the number of witnesses x that satisfy ψ is congruent to i modulo m, is studied.

Query evaluation algorithm is efficient if the update time is either constant or at most polylogarithmic ($\log^c n$) in the size of the database. As a consequence, efficient query evaluation in the dynamic setting is only conceivable if the static problem (i.e., the setting without database updates) can be solved for Boolean queries in linear or pseudo-linear ($n^{1+\varepsilon}$) time. Since this is not always possible, a short overview on known results about first-order query evaluation on static database is provided.

The problem of deciding whether a given database D satisfies a FO-sentence Q is $\mathsf{AW}[*]$-complete (parameterised by $\|Q\|$) and it is therefore generally believed that the evaluation problem cannot be solved in time $f(\|Q\|)\|D\|^c$ for any computable f and constant c (here, $\|Q\|$ and $\|D\|$ denote the size of the query and the database, respectively). For this reason, a long line of research focused on increasing classes of sparse instances ranging from databases of *bounded degree* [89] (where every domain element occurs only in a constant number of tuples in the database) to classes that

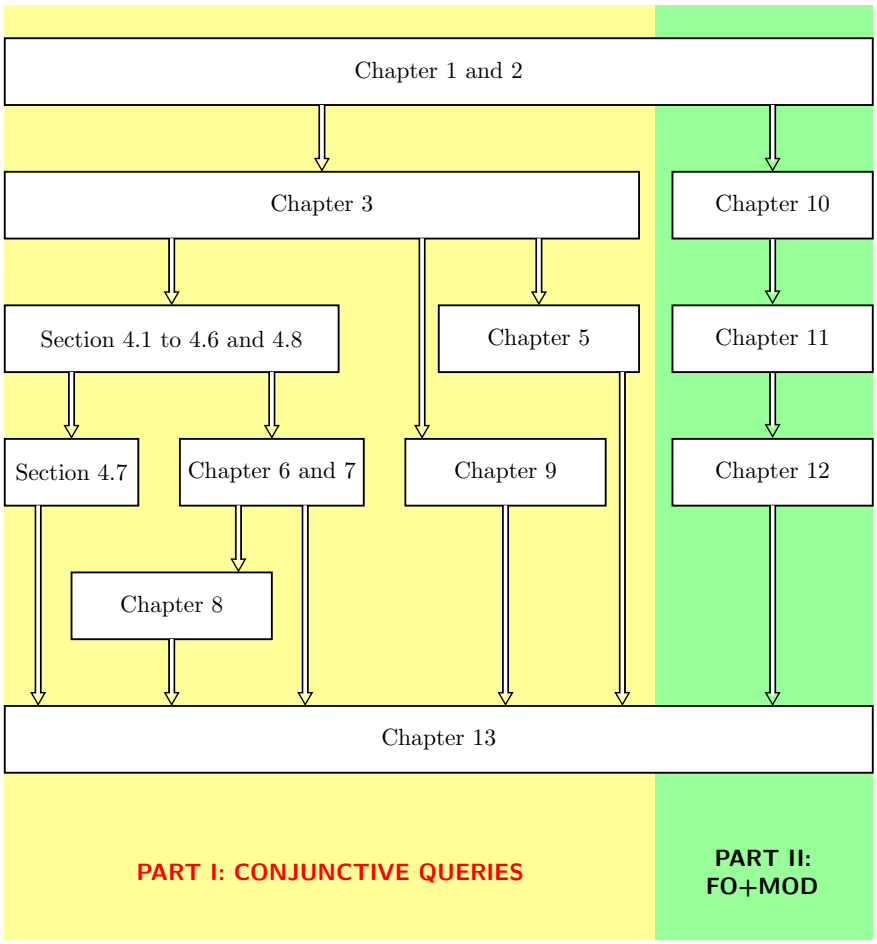
are *nowhere dense* [49]. In particular, Boolean first-order queries can be evaluated on classes of databases of bounded degree in linear time $f(\|Q\|)\|D\|$, where the constant factor $f(\|Q\|)$ is 3-fold exponential in $\|Q\|$ [89, 44], and Frick and Grohe [44] showed that the 3-fold exponential blow-up in terms of the query size is unavoidable assuming $\mathsf{FPT} \neq \mathsf{AW}[*]$.

Durand and Grandjean [36] and Kazana and Segoufin [60] considered the task of enumerating the result of a k-ary first-order query on bounded degree databases and showed that after a linear time preprocessing phase the query result can be enumerated with constant delay. This result was later extended to classes of databases of bounded expansion [61]. Kazana and Segoufin [61] also showed that counting the number of result tuples of a k-ary first-order query on databases of bounded expansion (and hence also on databases of bounded degree) can be done in time $f(\|Q\|)\|D\|$. Segoufin and Vigny [93] proved an analogous result for classes of locally bounded expansion and pseudo-linear time $f(\|Q\|)\|D\|^{1+\varepsilon}$, and they also presented an algorithm for enumerating the query result with constant delay after pseudo-linear time preprocessing. These results were recently generalised to all nowhere dense classes of databases by Grohe and Schweikardt [50] and Schweikardt, Segoufin and Vigny [87]. Durand, Schweikardt and Segoufin [38] obtained analogous results for classes of databases of low degree (i.e., degree at most $\|D\|^{o(1)}$).

1.3. Recommended reading order

Figure 1.1 shows a diagramm that illustrates in which sequence the reader can read this thesis.

Figure 1.1.: Recommended reading order.



2. Preliminaries

We write $\mathbb{N}$ for the set of non-negative integers and let $\mathbb{N}_{\geqslant 1} := \mathbb{N} \setminus \{0\}$ and $[n] := \{1, \ldots, n\}$ for all $n \in \mathbb{N}_{\geqslant 1}$. By 2^S we denote the power set of a set S. For a partial function f we write $\mathrm{dom}(f)$ and $\mathrm{codom}(f)$ for the domain and the codomain of f, respectively. When writing $poly(n)$, we mean $n^{O(1)}$ and when writing $\exp(n)$ we mean $2^{poly(n)}$.

Let A be a set and let $<$ be an order on A and let $\overline{a}, \overline{b} \in A^k$ be two tuples of length k with $\overline{a} = (a_1, \ldots, a_k)$ and $\overline{b} = (b_1, \ldots, b_k)$. A tuple $\overline{a}$ is lexicographically smaller than $\overline{b}$, symb. $\overline{a} <_{\mathrm{lex}} \overline{b}$, if there is a $j \in [k]$ such that $b_j < a_j$ and for all $i < j$ we have $b_j = a_j$. We write $\overline{a} \leqslant_{\mathrm{lex}} \overline{b}$ if $\overline{a} <_{\mathrm{lex}} \overline{b}$ or $\overline{a} = \overline{b}$.

Let G be a graph. We will usually write $V(G)$ to denote the vertex set of G and $E(G)$ to denote the edge set of G. Let T be a tree.

Principle of Inclusion and Exclusion. In this thesis, one often has to compute the number of elements in the union of sets. To determine these cardinalities, we use the *principle of inclusion and exclusion* (see [25] for the statement and a proof). Let X be a set and $(A_1, \ldots, A_n)$ be a family of subsets of X. Then the number of elements of X which lie in none of the subsets A_i is

$$\sum_{\emptyset \neq \mathcal{I} \subseteq \{1,\ldots,n\}} (-1)^{|\mathcal{I}|} \left| \bigcap_{i \in \mathcal{I}} A_i \right| + |X| \,. \tag{2.1}$$

Using (2.1) and setting $X = A_1 \cup \cdots \cup A_n$, we obtain the following

$$\left| \bigcup_{i=1}^{n} A_i \right| = \sum_{\emptyset \neq \mathcal{I} \subseteq \{1,\ldots,n\}} (-1)^{|\mathcal{I}|+1} \left| \bigcap_{i \in \mathcal{I}} A_i \right| \tag{2.2}$$

Databases. We fix a countably infinite set **dom**, the *domain* of potential database entries. Elements in **dom** are called *constants*. A *schema* is a finite set σ of relation symbols, where each $R \in \sigma$ is equipped with a fixed *arity* $\mathrm{ar}(R) \in \mathbb{N}$. Let us fix a schema $\sigma = \{R_1, \ldots, R_s\}$ and let $r_i := \mathrm{ar}(R_i)$ for $i \in [s]$. A *database* D of schema σ (σ-db for short) is of the form $D = (R_1^D, \ldots, R_s^D)$ where each R_i^D is a finite subset of $\mathbf{dom}^{r_i}$. The *active domain* $\mathrm{adom}(D)$ of D is the smallest subset A of **dom** such that $R_i^D \subseteq A^{r_i}$ for all $i \in [s]$.

Updates. We allow to update a given database of schema σ by inserting or deleting tuples as follows. An *insertion* command is of the form

$$\mathsf{insert}\ R(a_1, \ldots, a_r)$$

for $R \in \sigma$, $r = \text{ar}(R)$ and $a_1, \ldots, a_r \in \mathbf{dom}$. When applied to a σ-db D, the result is the updated σ-db D' with $R^{D'} := R^D \cup \{(a_1, \ldots, a_r)\}$ and $S^{D'} := S^D$ for all $S \in \sigma \setminus \{R\}$. A *deletion* command is of the form

$$\mathsf{delete}\ R(a_1, \ldots, a_r)$$

for $R \in \sigma$, $r = \text{ar}(R)$ and $a_1, \ldots, a_r \in \mathbf{dom}$. When applied to a σ-db D, the result is the updated σ-db D' with $R^{D'} := R^D \setminus \{(a_1, \ldots, a_r)\}$ and $S^{D'} := S^D$ for all $S \in \sigma \setminus \{R\}$. Note that both types of commands may change the database's active domain.

Sizes and Cardinalities. The *cardinality* $|D|$ of a σ-db D is defined as the number of tuples stored in D, i.e., $|D| := \sum_{R \in \sigma} |R^D|$. The *size* $\|D\|$ of D is defined as $|\sigma| + |\text{adom}(D)| + \sum_{R \in \sigma} \text{ar}(R) \cdot |R^D|$ and corresponds to the size of a reasonable encoding of D. We will often write n to denote the cardinality $|\text{adom}(D)|$ of D's active domain.

Dynamic Algorithms for Query Evaluation. The algorithms take as input a k-ary query $Q(x_1, \ldots, x_k)$ and a σ-db D_0. For all query evaluation problems considered in this thesis, we aim at routines **preprocess**, **update** and **init** which achieve the following:

- upon input of $Q(x_1, \ldots, x_k)$ and D_0, **preprocess** builds a data structure `D` which represents D_0 and which is designed in such a way that it supports efficient evaluation of Q on D_0,
- upon input of a command $\mathsf{update}\ R(a_1, \ldots, a_r)$ (with $\mathsf{update} \in \{\mathsf{insert}, \mathsf{delete}\}$), calling **update** modifies the data structure `D` such that it represents the updated database D and
- upon input of $Q(x_1, \ldots, x_k)$, we denote with **init** the particular case of the routine **preprocess** upon input of the query $Q(x_1, \ldots, x_k)$ and the *empty* database $D_\emptyset$, where $R^{D_\emptyset} = \emptyset$ for all $R \in \sigma$.

The *preprocessing time* t_p is the time used for performing **preprocess**; the *update time* t_u is the time used for performing an **update**; the *initialisation time* t_i is the time used for performing **init**. In all dynamic algorithms presented in this thesis, the **preprocess** routine for input of $Q(x_1, \ldots, x_k)$ and D_0 will carry out the **init** routine for $Q(x_1, \ldots, x_k)$ and then perform a sequence of $|D_0|$ update operations to insert all the tuples of D_0 into the data structure. Consequently, $t_p = t_i + |D_0| \cdot t_u$. Thus, it is sufficient to consider the **update** and the **init** routine instead of a **preprocess** routine, if $t_p = t_i + |D_0| \cdot t_u$.

An enumeration routine **enumerate** is a routine that invokes an enumeration of the elements of a set M and afterwards outputs an end-of-enumeration message `EOE`. The delay of **enumerate** is defined as the maximum time used during a call of **enumerate**

- until the output of the first element in M (or the end-of-enumeration message `EOE`, if $M = \emptyset$),
- between the output of two consecutive elements in M and

- between the output of the last element in M and the end-of-enumeration message `EOE`.

In the following, D will always denote the database that is currently represented by the data structure `D`. To solve the *enumeration problem under updates*, apart from the routines **preprocess** and **update**, we aim at a routine **enumerate** such that calling **enumerate** invokes an enumeration of all tuples, without repetition, that belong to the query result $Q(D)$. The *delay* t_d is the delay of **enumerate**.

To *test* if a given tuple belongs to the query result, we aim at a routine **test** which upon input of a tuple $\overline{a} \in \mathbf{dom}^k$ checks whether $\overline{a} \in Q(D)$. The *testing time* t_t is the time used for performing a **test**.

To solve the *counting problem under updates*, we aim at a routine **count** which outputs the cardinality $|Q(D)|$ of the query result. The *counting time* t_c is the time used for performing a **count**.

To *answer* a *Boolean* query under updates, we aim at a routine **answer** that produces the answer `yes` or `no` of Q on D. The *answer time* t_{ans} is the time used for performing **answer**.

To *output the jth solution*, we aim at a routine **jth** which outputs upon input of a number $j \in \mathbb{N}_{\geqslant 1}$ the jth tuple the **enumerate** routine would output if j is smaller or equal to the number of tuples in the result. The *jth time* t_j is the time used for performing **jth**.

To *enumerate the difference*, we aim at a routine **diff** which does the following. For a predefined σ-db D^- and a σ-db D where D is a database obtained from D^- after a couple of update steps, **diff** shall enumerate the sets $Q(D^-) \setminus Q(D$ and $Q(D) \setminus Q(D^-)$. The *diff-delay time* t_{di} is the delay for the routine **diff**.

To *prepare the learning of the result set*, we aim at a routine **learn** which prepares the algorithm for learning the query result as the training set in the polynomial regression model. The *learning time* t_l is the time for the routine **learn**.

Whenever speaking of a *dynamic algorithm*, we mean an algorithm that has routines **preprocess** and **update** and, depending on the problem at hand, at least one of the routines **enumerate**, **count**, **test**, **jth**, **diff**, **learn** and **answer**.

Machine Model. Following [32], we use Random Access Machines (RAMs) with $O(\log n)$ word-size and a uniform cost measure to analyse our algorithms. We will assume that the RAM's memory is initialised to 0. In particular, if an algorithm uses an array, we will assume that all array entries are initialised to 0 and this initialisation comes at no cost (in real-world computers this can be achieved by using the *lazy array initialisation technique*, cf. e.g. [77]). A further assumption is that for every fixed dimension $k \in \mathbb{N}_{\geqslant 1}$ we have an unbounded number of k-ary arrays `A` available such that for given $(n_1, \ldots, n_k) \in \mathbb{N}^k$ the entry $\mathtt{A}[n_1, \ldots, n_k]$ at position $(n_1, \ldots, n_k)$ can be accessed in constant time.[1]

Throughout this thesis, we often adopt the view of *data complexity* and suppress factors that may depend on the query Q or the degree bound d (the degree bound

[1] While this can be accomplished in the RAM-model, for an implementation on real-world computers one would probably have to resort to replacing our use of arrays by using suitable designed hash functions.

is only relevant for Part II) but not on the database D. E.g., "linear preprocessing time" means $t_p \leqslant g(Q,d) \cdot \|D\|$ and "constant update time" means $t_u \leqslant g(Q,d)$ for a function g with codomain $\mathbb{N}$.

The yield command. To keep algorithms simple and to simply analyse their running time we will often use the `yield` operator in our algorithms. This operator is also used in popular programming languages such as Python. The yield operator will be used in functions. As a consequence, these functions become iterable and an iteration over the functions works as follows. When we iterate over a function, the function starts in the first iteration and runs until it reaches the yield operator and then it outputs the value of the **yield** command. If the next value for the function is requested, the function continues from the point it stopped in the previous iteration and continues until it reaches the next yield and outputs the value of the yield operation. An example is given by the following Python code.

```
def f():
    for i in [1,2,3]:                    // Iterate for i in {1,2,3}
        yield i*i
        print("Output for " + str(i))

for l in f():
    print(l)
    print("End")
```

With the yield operator, the function `f` is iterable. When we start with the for loop `l in f()` the function runs until `yield 1*1` is called and we receive `1` as output. Then we continue with the block of the for-loop `l in f()` and print `End` and then we continue with `f()` and print `Output for 1` and then stop at `yield 2*2` and so on. The for loop is finished when the execution of `f()` is finished, i.e., `Output for 3` will be printed at last. The output of the example is

```
1
End
Output for 1
4
End
Output for 2
9
End
Output for 3
```

The running time between two iteration steps of a function with yield is the maximum of the following running times:

- the time until we reach the first yield and
- the time between two yield operations and

- the time between a yield operation and the end of the function.

The list data structure. In the whole thesis, we consider the following dynamic data structure that represents a list which can be initialised and updated in constant time. For each list we have a type $\mathtt{T}$ which represents the set of possible items. We write $\mathcal{L} = \langle b_1, \cdots, b_\ell \rangle$ to denote a list of type $\mathtt{T}$ with ℓ elements $b_1, \ldots, b_\ell \in \mathtt{T}$. For example, $\langle 4, 6, 7 \rangle$ is a list of type $\mathbb{N}$ with the elements $4, 6, 7$. For a list $\mathcal{L} = \langle b_1, \cdots, b_\ell \rangle$ and an element $a \in \mathtt{T}$, we write $a \in \mathcal{L}$ if $a \in \{b_1, \ldots, b_\ell\}$ and $a \notin \mathcal{L}$ if $a \notin \{b_1, \ldots, b_\ell\}$.

We will first describe the data structure where an element $a \in \mathtt{T}$ can appear at most once in the list. Afterwards, we extend the data structure such that we can maintain a list where an element can appear multiple in the list.

The data structure for a list consists of an array $\mathtt{A}$, a pointer *start* and a set of items which represents elements in the list. For every item i in the list we store the value $i.a$ which represents the corresponding element of $\mathtt{T}$ and a pointer $i.\mathtt{next}$ that points to the successor element in the list, if it exists and to $\mathtt{nil}$ otherwise and a pointer $i.\mathtt{prev}$ that points to the previous element in the list, if it exists and to $\mathtt{nil}$ otherwise. The $\mathtt{next}$ and the $\mathtt{prev}$ pointer will help us to quickly insert and remove items in the list or $\mathtt{nil}$ if it is the empty list. The array $\mathtt{A}$ maps an element in $\mathtt{T}$ to the item in the list if it is part of the list and to $\mathtt{nil}$ otherwise. The pointer *start* points to the first item in the list.

For example, the data structure for the list $\langle 4, 6, 7 \rangle$ (that is of type $\mathbb{N}$) is the following. There are three items i_1, i_2, i_3 with $i_1.a = 4$ and $i_2.a = 6$ and $i_3.a = 7$ and $i_j.\mathtt{next} = i_{j+1}$ for all $j \in \{1, 2\}$ and $i_j.\mathtt{prev} = i_{j-1}$ for all $j \in \{2, 3\}$ and $i_1.\mathtt{prev} = i_3.\mathtt{next} = \mathtt{nil}$ and $\mathit{start} = i_1$. We have $\mathtt{A}[4] = i_1$, $\mathtt{A}[6] = i_2$, $\mathtt{A}[7] = i_3$ and $\mathtt{A}[n] = 0$ for $n \in \mathbb{N} \setminus \{4, 6, 7\}$.

To *initialise* the data structure for the empty list, we initialise the array $\mathtt{A}$ (note that by definition we can assume that $\mathtt{A}[a] = 0$ for all $a \in \mathtt{T}$ after the initialisation) and set the *start* pointer to $\mathtt{nil}$. This can be done in $O(1)$.

The aim is for a list $\mathcal{L}$ of type $\mathtt{T}$ to support two update operations $\mathsf{insert}(b, a)$ and $\mathsf{remove}(b)$ where $b \in \mathcal{L} \dot\cup \{\mathtt{nil}\}$ and $a \in \mathtt{T}$. On input of an update operation $\mathsf{insert}(b_k, a)$ with $b_k \neq \mathtt{nil}$ we insert a as the successor b_k to the list if $a \notin \mathcal{L}$, i.e., for the list $\mathcal{L} = \langle b_1, \ldots, b_\ell \rangle$ where $k \leqslant \ell$, the result after the update is $\langle b_1, \ldots, b_k, a, b_{k+1}, \ldots, b_\ell \rangle$. If $b_k = \mathtt{nil}$, we insert a as the first element to the list, i.e., if the list $\mathcal{L} = \langle b_1, \ldots, b_\ell \rangle$ receives the update command $\mathsf{insert}(\mathtt{nil}, a)$, then the resulting list is $\langle a, b_1, \ldots, b_\ell \rangle$. We will often use a variant of the insertion operation $\mathsf{insert}(a)$ where no previous element is specified. Then we will add the new item as the last element in the list, i.e., $\mathsf{insert}(a)$ calls $\mathsf{insert}(b_\ell, a)$ where b_ℓ is the last element of $\mathcal{L}$ or $b_\ell = \mathtt{nil}$ if $\mathcal{L}$ is empty. If $\mathcal{L} = \langle b_1, \ldots, b_\ell \rangle$ receives the update operation $\mathsf{remove}(b_k)$ for $k \leqslant \ell$, we remove the item with value b_k from the list, i.e., we obtain the list $\langle b_1, \ldots, b_{k-1}, b_{k+1}, \ldots b_\ell \rangle$. The update operations will be implement as follows.

If the list $\mathcal{L} = \langle b_1, \ldots, b_\ell \rangle$ receives the update $\mathsf{insert}(b_k, a)$ and we implement the insertion algorithm as follows. If $b_k \neq \mathtt{nil}$, we let i^- be the be the list item representing b_k and i^+ be $i^-.\mathtt{next}$. Note that i^+ is $\mathtt{nil}$ if b_k is the last element in the list. We create a new list item i with $i.a = a$, $i.\mathtt{prev} = i^-$ and $i.\mathtt{next} = i^+$. We will also set $i^-.\mathtt{next}$ to i and, if $i^+ \neq \mathtt{nil}$, the value $i^+.\mathtt{prev}$ to i. If we have the case that $b_k = \mathtt{nil}$, we let i^+ be the value of the *start* pointer and create a new item i with $i.a = a$, $i.\mathtt{prev} = \mathtt{nil}$

and $i.\texttt{next} = i^+$. We will also set $i^+.\texttt{prev}$ to i if $i^+ \neq \texttt{nil}$. Moreover, we set $\texttt{A}[a]$ to the new item i in all cases. Note that we obtain a data structure for the list with the new element is inserted as above.

If the list $\mathcal{L} = \langle b_1, \ldots, b_\ell \rangle$ receives the update command $\mathsf{remove}(b_k)$, let i be the item which represents b_k and let i^- be $i.\texttt{prev}$ and i^+ be $i.\texttt{next}$. If i^- is not $\texttt{nil}$, we set $i^-.\texttt{next}$ to i^+ and otherwise *start* to i^+. If i^+ is not $\texttt{nil}$, we set $i^+.\texttt{prev}$ to i^-. Furthermore, we set $i.\texttt{prev} = i.\texttt{next} = \texttt{nil}$, and we set $\texttt{A}[a]$ to 0. Note that we obtain a data structure for the list where b_k is removed from the list as described above.

Since we have a constant number of "setting pointer to items" and lookup, insert and remove commands on $\texttt{A}$ during both update commands, an update on the list takes time $O(1)$.

The data structure also supports a **lookup**, an **empty** and an **enumeration** routine. The **lookup** routine $\mathsf{lookup}(a)$ receives as input an element $a \in \texttt{T}$ and outputs true if and only if $a \in \mathcal{L}$. To implement this operation, we simply output the result of testing $\texttt{A}[a] \neq 0$. This can be done in $O(1)$. The **empty** routine tests if the list is empty. This can be done in $O(1)$ by testing if *start* is set to $\texttt{nil}$. The **enumerate** routine enumerates the elements in the list. This can be implemented as follows. We let i be the item that is pointed from *start* (if *start* = $\texttt{nil}$, we output $\texttt{EOE}$ and terminate the enumerate operation). While $i \neq \texttt{nil}$, we yield $i.a$ and set i to $i.\texttt{next}$. Afterwards, we output $\texttt{EOE}$ and terminate the enumerate operation. The delay during the enumeration is $O(1)$.

To enrich that data structure for maintaining lists with multiple entries we use the following modifications. Instead of storing a single item in $\texttt{A}[a]$, we store in $\texttt{A}[a]$ a pointer to a list $\mathcal{L}_a$ of elements that consists of items in $\mathcal{L}$ whose values are a. Note that since the pointer to an item in $\mathcal{L}$ is unique, the elements in the list $\mathcal{L}_a$ appears at most once and we can use the data structure described above for $\mathcal{L}_a$. Furthermore, we prune in the insertion procedure the condition that $a \notin \mathcal{L}$ since an element can multiple appear in $\mathcal{L}$. When inserting an element a to the list $\mathcal{L}$ we add the new item to the list $\mathcal{L}_a$ (instead of setting $\texttt{A}[a]$ to the item) and if we remove the item b_k from the list $\mathcal{L}$ we remove the corresponding item from the list $\mathcal{L}_{i.\texttt{value}}$ (instead of setting $\texttt{A}[i.\texttt{value}] = 0$). To test if $a \in \mathcal{L}$ for the **lookup** routine, we test if the list in $\texttt{A}[a]$ is not empty and we do not have $\texttt{A}[a] = 0$. It is easy to verify that inserting and removing an element to/from $\mathcal{L}$ and the **lookup** routine can be done in $O(1)$.

AVL-trees for dynamic list representations. In this thesis we will often use list representations for which we have to find the kth item in the list efficiently. To realise this, we will use the idea of Adelson-Velskii and Landis [3] that allows us to design the following data structure.

Theorem 2.1 ([3],[65]). *There is a data structure that represents an arbitrary list of size N and allows the following operations in time $O(\log(N))$.*

1. *Upon input of a key, find the item in the list with the corresponding key,*
2. *upon input a number k, find the item in the kth position in the list,*
3. *insert an item (at a specific place) and*

4. delete a specified item.

We give an idea how the data structure in Theorem 2.1 works, see [65] for details. Additionally to a list we use a balanced binary tree. A balanced binary tree is a binary tree where for every node the following holds. The difference between the hight of the left child and the hight of the right node lies between -1 and 1. This gives a guarantee that the hight of such a tree is in $O(\log(N))$ where N is the size of the list.

For every node, we store two pointer `LLINK` and `RLINK` that stores a pointer to the left and the right child, respectively. Additionally, we store for every node the fields:

- `KEY`: that represents a value of a list if the node is a leaf,
- `B`: that represents the balance factor of the node, i.e., the number of the height of the left child minus the number of the height of the right child,
- `RANK`: one plus the number of leafs in the left subtree.

The field `RANK` will help us to find the kth element in the list. In a balanced tree the field `B` for every node must be -1, 0 or $+1$. After inserting or deleting an element to/from the list, the tree may not be balanced. To correct this, the algorithm makes some rotation on the tree (see Figure 2.1 and 2.2 for the rotations). In these figures, there are the two cases of an unbalanced trees shown. The red numbers show the balance factor of a node and the blue triangles depict that there is a subtree.

A detailed description of the mentioned algorithms is given in [65].

Figure 2.1.: Single-rotation. T_1 and T_2 are subtrees of height h and T_3 is a subtree of height $h+1$.

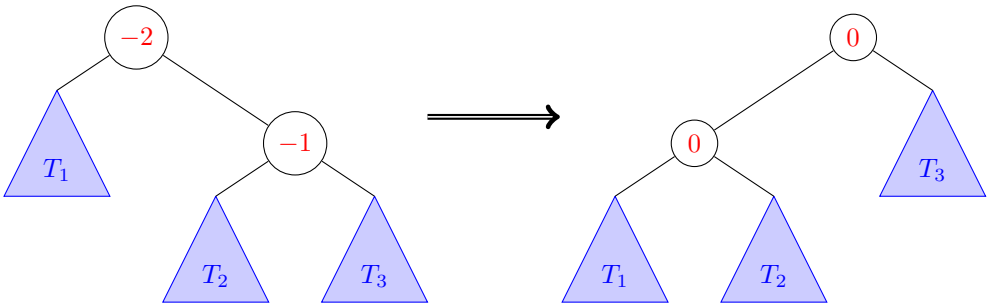
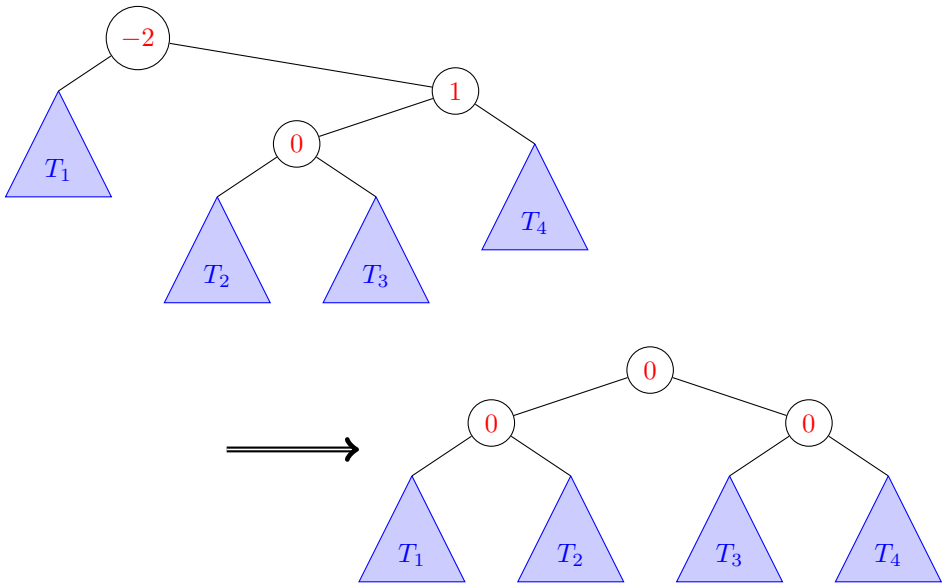


Figure 2.2.: Double rotation. Each subtree T_1, T_2, T_3 and T_4 has height h.



Part I.

Conjunctive Queries under Updates

3. Main Theorems and Organisation of Part I

This chapter starts with a definition of syntax and semantics of conjunctive queries. Then the main results of this part are given and an organization of the first part.

3.1. An introduction to conjunctive queries

We fix a countably infinite set **var** of *variables*. An *atomic query* (for short: *atom*) ψ of schema σ is of the form $Ru_1 \cdots u_r$ with $R \in \sigma$, $r = \mathrm{ar}(R)$ and $u_1, \ldots, u_r \in \mathbf{var} \cup \mathbf{dom}$. A *conjunctive query* (CQ, for short) of schema σ is of the form

$$\{(x_1, \ldots, x_k, b_1, \ldots, b_\ell) \ : \ \exists y_1 \cdots \exists y_\ell \ (\psi_1 \wedge \cdots \wedge \psi_d)\} \tag{3.1}$$

where $k, \ell \in \mathbb{N}$, $d \in \mathbb{N}_{\geqslant 1}$, ψ_j is an atomic query of schema σ for every $j \in [d]$ and $x_1, \ldots, x_k, y_1, \ldots, y_\ell$ are pairwise distinct elements in **var** and $b_1, \ldots, b_\ell \in \mathbf{dom}$. A formula of the form $\exists y_1 \cdots \exists y_\ell \ (\psi_1 \wedge \cdots \wedge \psi_d)$ is called conjunctive formula. *Join queries* are quantifier-free CQs, i.e., CQs of the form

$$\{(x_1, \ldots, x_k, b_1, \ldots, b_\ell) \ : \ (\psi_1 \wedge \cdots \wedge \psi_d)\}.$$

A CQ is called *self-join free* (or *non-repeating* or *simple*) if no relation symbol occurs more than once in the query. For a CQ Q of the form (3.1) we let $\mathrm{vars}(Q)$ ($\mathrm{cons}(Q)$) be the set of all variables (constants) occurring in Q, and we let $\mathrm{free}(Q) := \mathrm{vars}(Q) \setminus \{y_1, \ldots, y_\ell\} = \{x_1, \ldots, x_k\}$ be the set of *free* variables. For an atom ψ let $\mathrm{vars}(\psi)$ ($\mathrm{cons}(\psi)$) be the set of all variables (constants) occurring in Q. For every variable $x \in \mathrm{vars}(Q)$ we let $\mathrm{atoms}(x)$ be the set of all atoms ψ_j of Q such that $x \in \mathrm{vars}(\psi_j)$.

The semantics of CQs are defined as usual: A *valuation* is a mapping $\beta : \mathrm{vars}(Q) \cup \mathbf{dom} \to \mathbf{dom}$ with $\beta(a) = a$ for every $a \in \mathbf{dom}$. For every atom $\psi = Ru_1 \cdots u_r$ in Q we write $(D, \beta) \models \psi$ to denote that $(\beta(u_1), \ldots, \beta(u_r)) \in R^D$. A valuation β is a *homomorphism* from Q to a σ-db D if for every atom ψ in Q we have $(D, \beta) \models \psi$. We sometimes write $(D, \beta) \models Q$ to indicate that β is a homomorphism from Q to D. The *query result* $Q(D)$ of a k-ary CQ $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ on a σ-db D is defined as the set $\{\, (\beta(x_1), \ldots, \beta(x_k), b_1, \ldots, b_\ell) \ : \ \beta \text{ is a homomorphism from } Q \text{ to } D \,\}$. Clearly, $Q(D) \subseteq \mathrm{adom}(D)^{k+\ell}$. For a tuple $\overline{a} \in \mathbf{dom}^k$ we sometimes write $D \models Q[\overline{a}]$ to indicate that there is a homomorphism $\beta : Q \to D$ with $\overline{a} = (\beta(x_1), \ldots, \beta(x_k), b_1, \ldots, b_\ell)$. We sometimes write $(D, \alpha) \models \varphi$ for a conjunctive formula φ to indicate that $(D, \beta) \models Q$ for $Q := \{(x_1, \ldots, x_k, b_1, \ldots, b_\ell) \ : \ \varphi\}$ where $\{x_1, \ldots, x_k\}$ is the set of free variables

in φ, i.e., the set of variables occurring in φ that are not quantified and $\{b_1, \ldots, b_\ell\}$ the set of constants that appear in φ.

A *Boolean* CQ is a CQ Q with $\text{free}(Q) = \emptyset$. As usual, for Boolean CQs Q we will write $Q(D) = \texttt{yes}$ instead of $Q(D) \neq \emptyset$, and $Q(D) = \texttt{no}$ instead of $Q(D) = \emptyset$.

Two k-ary queries $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ and $Q'(x'_1, \ldots, x'_k, b_1, \ldots, b_\ell)$ are *equivalent* ($Q \equiv Q'$, for short) if $Q(D) = Q'(D)$ for every σ-db D.

The *size* $\|Q\|$ of a CQ Q is defined as the length of Q when viewed as a word over the alphabet $\sigma \cup \mathbf{var} \cup \mathbf{dom} \cup \{\exists, \wedge, (,), \{, \}\}$. For a k-ary CQ $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ and a σ-db D, the *cardinality of the query result* is the number $|Q(D)|$ of tuples in $Q(D)$.

The *artiy of a CQ* $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is the number $k + \ell$.

We write *poly*(Q) as an abbreviation for *poly*$(\|Q\|)$ and we write $\exp(Q)$ as an abbreviation for $\mathsf{expr}(\|Q\|)$.

UCQs. A k-ary *union of conjunctive queries* (k-ary UCQ) is of the form $Q_1(\overline{u}_1) \cup \cdots \cup Q_d(\overline{u}_d)$ where $d > 1$ and $Q_i(\overline{u}_i)$ is a k-ary CQ of schema σ for every $i \in [d]$. The query result of such a k-ary UCQ Q on a σ-db D is $Q(D) := \bigcup_{i=1}^{d} Q_i(D)$. For a k-ary query Q we write $\text{vars}(Q)$ (and $\text{cons}(Q)$) to denote the set of all variables (and constants) that occur in Q. Clearly, $Q(D) \subseteq (\text{adom}(Q) \cup \text{cons}(Q))^k$.

3.2. Main theorems and organisation

The aim of the thesis' first part is to investigate the following problem. For which class of conjunctive queries, we can construct a data structure that represents the result set on a database that is tractable under updates and for which we can design fast enumeration, answering, testing, difference, jth routines. It turns out that the class of conjunctive queries for which we can realise this is the class of q-hierarchical conjunctive queries, which are defined as follows.

Definition 3.1. *A CQ Q is* q-hierarchical *if for any two variables $x, y \in \text{vars}(Q)$ the following is satisfied:*

(i) $\text{atoms}(x) \subseteq \text{atoms}(y)$ *or* $\text{atoms}(x) \supseteq \text{atoms}(y)$ *or* $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$, *and*

(ii) if $\text{atoms}(x) \subsetneq \text{atoms}(y)$ *and* $x \in \text{free}(Q)$, *then* $y \in \text{free}(Q)$.

Minimal examples for queries that are not q-hierarchical are

$$Q_{S\text{-}E\text{-}T} \quad := \quad \{(x, y, z) \ : \ (Sx \wedge Exy \wedge Ty)\} \tag{3.2}$$

and

$$Q_{E\text{-}T} \quad := \quad \{(x) \ : \ \exists y\,(Exy \wedge Ty)\} \tag{3.3}$$

because they do not satisfy condition (i) and (ii), respectively. Note that some variations of $Q_{E\text{-}T}$ such as the join query $\{(x, y) \ : \ (Exy \wedge Ty)\}$, the Boolean query $\{() \ : \ \exists x \exists y\,(Exy \wedge Ty)\}$ and the query $\{(y) \ : \ \exists x\,(Exy \wedge Ty)\}$ are q-hierarchical.

The notion of *q-hierarchical* conjunctive queries is related to the *hierarchical* property that has already played a central role for efficient query evaluation in various contexts. It has been introduced by Dalvi and Suciu in [34] to characterise the Boolean CQs that can be answered in polynomial time on probabilistic databases. They obtained a dichotomy stating for self-join free queries that the complexity of query evaluation on probabilistic databases is in PTIME for hierarchical queries and #P-complete for non-hierarchical queries. Fink and Olteanu [42] generalised the notion and the dichotomy result to non-Boolean queries and to queries using negation. In the different context of query evaluation on massively parallel architectures, Koutris and Suciu [69] considered hierarchical join queries and singled out a subclass of so-called tall-flat queries as exactly those queries that can be computed with only one broadcast step in their *Massively Parallel* model of query evaluation. For further information on the various uses of the hierarchical property the author refers the reader to [42].

The definition of hierarchical queries relies on the following notion. Consider a CQ Q of the form (3.1). Dalvi and Suciu [34] call a Boolean CQ Q *hierarchical* iff the condition

$$(*): \quad \text{atoms}(x) \subseteq \text{atoms}(y) \quad \text{or} \quad \text{atoms}(x) \supseteq \text{atoms}(y) \quad \text{or} \quad \text{atoms}(x) \cap \text{atoms}(y) = \emptyset$$

is satisfied by all variables $x, y \in \text{vars}(Q)$. An example for a hierarchical Boolean CQ is $\{() : \exists x \exists y \exists z \exists y' \exists z' \, (Rxyz \wedge Rxyz' \wedge Exy \wedge Exy')\}$.

In [69], Koutris and Suciu transferred the notion to join queries Q, which they call hierarchical iff condition $(*)$ is satisfied by all variables $x, y \in \text{vars}(Q)$. In [42], Fink and Olteanu introduced a slightly different notion for a more general class of queries. Translated into the setting of CQs, their notion (only) requires that condition $(*)$ is satisfied by all *quantified* variables, i.e., variables $x, y \in \text{vars}(Q) \setminus \text{free}(Q)$. Obviously, both notions coincide on *Boolean* CQs, but on *join queries* Koutris and Suciu's notion is more restrictive than Fink and Olteanu's notion (according to which *all* quantifier-free CQs are hierarchical). For example, the join query Q_{S-E-T} from Equation (3.2) is hierarchical w.r.t. Fink and Olteanu's notion, and non-hierarchical w.r.t. Koutris and Suciu's notion.

To ensure tractability of a conjunctive query in that setting, we will require that its quantifier-free part is hierarchical in Koutris and Suciu's notion and, additionally, the quantifiers respect the query's hierarchical form. Such queries are the *q-hierarchical* queries.

Note that a *Boolean* CQ is q-hierarchical if and only if it is hierarchical, and a join query is q-hierarchical if and only if it is hierarchical w.r.t. Koutris and Suciu's notion.

This notion of q-hierarchical queries is also related to the notion of factorized databases from [82]. Every query result of a q-hierarchical query can be identified as a factorized database but not every factorized database can be described by a q-hierarchical query. Moreover, factorized database have only been considered in the static setting, i.e., in the setting without updates.

In the first part of this thesis we will prove the following theorems for q-hierarchical CQs:

Theorem 3.2. *There is a dynamic algorithm that receives a Boolean q-hierarchical conjunctive query Q and a σ-db D_0, and computes within $t_i = poly(Q)$ initialisation time and $t_p = poly(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows to answer $Q(D)$ in time $t_{ans} = O(1)$ where D is the current database.*

For non-Boolean q-hierarchical CQs we receive the following theorem.

Theorem 3.3. *There is a dynamic algorithm that receives a non-Boolean q-hierarchical conjunctive query $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ and a σ-db D_0 and computes within $t_i = poly(Q)$ initialisation time and $t_p = poly(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows to*

(a) test for an input tuple $\overline{a} \in \mathbf{dom}^{k+\ell}$ if $\overline{a} \in Q(D)$ within time $t_t = poly(Q)$, and

(b) enumerate $Q(D)$ with delay $t_d = poly(Q)$, and

(c) enumerate the tuples that join and omit the result set $Q(D)$ after an arbitrary number of updates on a previous chosen state of the database received with delay $t_{di} = poly(Q)$ and

(d) compute the cardinality $|Q(D)|$ in time $t_c = O(1)$

where D is the current database.

The result from Theorem 3.2 and Theorem 3.3 (a) (b) and (d) are results from [16]. The proof of these theorems are different to the proofs in [16]. The result of Theorem 3.3 (c) is a new result.

A proof for Theorem 3.2 and Theorem 3.3 is given in Chapter 4 (except for part (d) of Theorem 3.3. This is proven in Section 6.5). The data structure used in these theorems are the same and it is described in Section 4.1. In Section 4.3 a proof for Theorem 3.2 is given. Then, we show how to use the data structure for the testing problem (see Section 4.4) and for the enumeration problem (see Section 4.6) and for the difference problem (see Section 4.7). It turns out that there are non q-hierarchical queries for which testing can be done efficiently under updates. This superclass of the q-hierarchical CQs are called t-hierarchical queries and are defined as follows.

Definition 3.4. *A CQ Q is* t-hierarchical *if the following is satisfied.*

(i) for all $x, y \in \text{vars}(Q) \setminus \text{free}(Q)$, we have

$\text{atoms}(x) \subseteq \text{atoms}(y)$ *or* $\text{atoms}(y) \subseteq \text{atoms}(x)$ *or* $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$, *and*

(ii) for all $x \in \text{free}(Q)$ and all $y \in \text{vars}(Q) \setminus \text{free}(Q)$, we have

$\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$ *or* $\text{atoms}(y) \subseteq \text{atoms}(x)$.

The queries $Q_{E\text{-}E\text{-}R} := \{ (x, y) : \exists v_1 \exists v_2 \exists v_3 (Exv_1 \wedge Eyv_2 \wedge Rxyv_3) \}$ and $Q_{S\text{-}E\text{-}T}$ from Equation (3.2) are examples of queries that are t-hierarchical but not q-hierarchical. In fact, the only difference between q-hierarchical CQs and t-hierarchical CQs relies on condition (i) in their definitions. While

$$\text{atoms}(x) \subseteq \text{atoms}(y) \quad \text{or} \quad \text{atoms}(x) \supseteq \text{atoms}(y) \quad \text{or} \quad \text{atoms}(x) \cap \text{atoms}(y) = \emptyset$$

holds for all $x, y \in \text{vars}(Q)$ for q-hierarchical queries, the fact has to hold only for $x, y \in \text{vars}(Q) \setminus \text{free}(Q)$. The second condition in Definition 3.1 (for q-hierarchical queries) is equivalent to the second condition in Definition 3.4 which follows from the following lemma.

Lemma 3.5. *For all CQs Q for which condition* (i) *from Definition 3.1 holds are the following two conditions equivalent:*

(a) For all $x, y \in \text{vars}(Q)$ the following holds: if $\text{atoms}(x) \subsetneq \text{atoms}(y)$ and $x \in \text{free}(Q)$, then $y \in \text{free}(Q)$.

(b) For all $x \in \text{free}(Q)$ and all $y \in \text{vars}(Q) \setminus \text{free}(Q)$, we have $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$ or $\text{atoms}(y) \subseteq \text{atoms}(x)$.

Proof. Condition (a) holds, if and only if

$$\begin{gathered} \text{for all } x, y \in \text{vars}(Q) \text{ is } x \notin \text{free}(Q) \text{ or } y \in \text{free}(Q) \\ \text{or it holds not } \text{atoms}(x) \subsetneq \text{atoms}(y). \end{gathered} \tag{3.4}$$

From condition (i) of Definition 3.1 follows that it holds not $\text{atoms}(x) \subsetneq \text{atoms}(y)$, if and only if $\text{atoms}(x) \supseteq \text{atoms}(y)$ or $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$. Thus, (3.4) is equivalent to

$$\begin{gathered} \text{for all } x, y \in \text{vars}(Q) \text{ is } x \notin \text{free}(Q) \text{ or } y \in \text{free}(Q) \\ \text{or it holds } \text{atoms}(x) \supseteq \text{atoms}(y) \text{ or } \text{atoms}(x) \cap \text{atoms}(y) = \emptyset \end{gathered} \tag{3.5}$$

It is straightforward to see that (3.5) and (b) are equivalent. □

We obtain the following theorem for the testing problem:

Theorem 3.6 ([21])**.** *There is a dynamic algorithm that receives a t-hierarchical k-ary CQ Q and a σ-db D_0, and computes within $t_p = \text{poly}(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = \text{poly}(Q)$ and allows to test for an input tuple $\overline{a} \in \mathbf{dom}^k$ if $\overline{a} \in Q(D)$ within time $t_t = \text{poly}(Q)$, where D is the current database.*

A proof for Theorem 3.6 is given in Chapter 5. Theorem 3.6 has already been published in [21] and the proof is based on the proof in [21].

In Chapter 6, we enrich q-hierarchical conjunctive queries to queries with aggregates.

In practice, aggregates can be used in every commercial implementation of SQL. Furthermore there are a lot of applications for queries with aggregates such as mobile computing [12], global information systems [71], stream data analysis [35], constraint databases [14]. That is the reason why lot of research has been done on evaluating

queries with aggregates [27, 52, 94, 31]. Moreover, aggregates can also be used to extend logics [56, 13, 40]. Only little is known about the enumeration problem for queries with aggregates in the static setting. In [10] it is shown how to enumerate the query result where the queries are factorized queries with aggregates that are very close to the q-hierarchical queries. In [10] the static setting has been considered.

In Chapter 6 we allow to operate over the result set with aggregates. In Section 6.1 there are examples for queries with aggregates and in Section 6.3 a formal definition of the syntax and semantics of such queries is given. It turns out that queries with aggregates are convenient to solve the counting problem for q-hierarchical conjunctive queries without aggregates, i.e., we can use queries with aggregates to prove Theorem 3.3(d) (see Section 6.5). In Section 6.6 and Section 6.7 we show how we can if the result of Theorem 3.3 from q-hierarchical conjunctive queries without aggregates to queries with aggregates and we obtain a proof for the following theorem:

Theorem 3.7. *There is a dynamic algorithm that receives a q-hierarchical CQ with aggregates Q with aggregation time t_a and a σ-db D_0, and computes within $t_i = O(1)$ initialisation time and $t_p = poly(Q)t_aO(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)t_a$ and allows to*

(a) enumerate $Q(D)$ with delay $t_d = poly(Q)$,

(b) test for an input tuple $\overline{a} \in \mathbf{dom}^k$ if $\overline{a} \in Q(D)$ within time $t_t = poly(Q)$,

(c) enumerate the tuples that join and omit the result set $Q(D)$ after an arbitrary number of updates on a previous chosen state the database received with delay $t_{di} = poly(Q)$ and

(d) compute the cardinality $|Q(D)|$ in time $t_c = O(1)$

where D is the current database.

This result is yet unpublished and a new result.

The aggregation time t_a of a query is the time it takes to update the values computed by the aggregation functions during an update step.

In Chapter 7 we show how to use the query result of a database query to prepare learning a polynomial function where the query result is taken as training data.

In the last years, the research interest for connecting database with machine learning has been growing [1].

Most machine learning systems and database systems do not support a connection between a database and the machine learning algorithm. To learn with data from a query result, one has to store the query result from a database system in a file that can be used as input for a machine learning algorithm. Such a data exchange is very expensive due to the repetition of data blocks e.g. in the result of join queries. That is the reason why it might be convenient to do some computation step of the machine learning algorithm in the database system, the so-called in-database learning. In-database learning was considered by Schleich and Olteanu [86] for factorized databases and Khamis et.al. [62] introduced a unified framework for in-database learning of

a class of statistical learning models in relation databases. In this thesis there is a in-database learning solution for regression models in relational databases on q-hierarchical queries under updates. Every time a tuple is inserted into or deleted from the database, we can appropriately modify the data structure of the in-database learning system.

Further work has been done in the task of Machine Learning and databases [64, 58, 84, 5].

We obtain the following new, unpublished, theorem.

Theorem 3.8. *For every $d \in \mathbb{N}_{\geqslant 1}$ there is a dynamic algorithm that receives a q-hierarchical conjunctive query Q and a σ-db D_0 and computes within $t_i = O(poly(Q)^{2d+1})$ initialisation time and $t_p = poly(Q)^{2d+1} \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)^{2d+1}$ and allows to prepare the learning of the parameters of a polynomial hypothesis function h_Θ of degree d in time $t_l = O(poly(Q)^{2d+1})$ where D is the current database.*

Afterwards we consider the question if there are more routines we can do, if we allow logarithmic update time. When allowing logarithmic update time, one can organize the data structure from Theorem 3.3, such that we can enumerate the tuples in lexicographical order. This means, if a tuple $\overline{a}$ will be enumerated before $\overline{b}$, then $\overline{a} <_{\text{lex}} \overline{b}$. Furthermore, one can upon input of a natural number output the jth tuple the enumeration routine would output and, upon input of a tuple in the result set, receive a natural number j such that the tuple is the jth tuple in the enumeration. This is stated in the next Theorem.

Theorem 3.9. *There is a dynamic algorithm that receives a q-hierarchical k-ary CQ with aggregates Q with aggregation time t_a and a σ-db D_0 of size $\|D_0\|$, and computes within preprocessing time $poly(Q) \cdot O(\|D_0\| \log(\|D_0\|))t_a$ a data structure that can be updated in time $poly(Q) \cdot O(\log(\|D\|))t_a$ and allows the following:*

(a) output the query result size $|Q(D)|$ in time $O(1)$ and

(b) enumerate the tuples in $Q(D)$ in lexicographical order with delay $poly(Q)$ and

(c) test for an input tuple $\overline{a} \in \mathbf{dom}^{k+\ell}$ if $\overline{a} \in Q(D)$ within time $t_t = poly(Q)$ and

(d) upon input of a tuple $\overline{b} \in \mathbf{dom}^{k+\ell}$ output the tuple

$$\max\left\{\overline{a} \in Q(D) \ : \ \overline{a} \leqslant \overline{b}\right\}$$

if it exists, or a `SmallerThanMinimum`-message otherwise in time $t_l = poly(Q)$ and

(e) upon input of an arbitrary number j, take time $poly(Q) \cdot O(\log(\|D\|))$ to immediately output the jth tuple that the enumeration procedure of (b) would output and

(f) upon input of an arbitrary tuple $\overline{a} \in Q(D)$, take time $poly(Q) \cdot O(\log(\|D\|))$ to immediately output the number j such that $\overline{a}$ is the jth tuple the enumeration procedure of (b) *would output,*

where D is the current database.

The result of this theorem is new and yet unpublished. To solve the routines in Part (a), (b), (c) and (d) we use a variant of the data structure from Theorem 3.3. The variant is described in Remark 4.9. Using this variant, we obtain that the query result will be enumerated in lexicographical order, when the enumeration algorithm from Theorem 3.3 is used (see Remark 4.15). Furthermore, we obtain the routine in Part (d) which is shown in Section 4.8.

For the routines in Part (e) and in Part (f) we enrich the data structure from Theorem 3.3 by another way. Note that, in particular, it is not necessary that the enumeration algorithm enumerates the tuples in lexicographical order for solving the jth problem and the jth-reverse problem. In Chapter 8, it is described how to enrich the data structure for the jth problem and the jth reverse problem and how these routines work.

To output the jth solution of the query result allows us to have random access on the query result by using any generation of integers. For the static setting Bagan, Durand, Grandjean and Olive [9] proved that for a FO query computing a random tuple of the query result can be done in average constant time after a linear preprocessing phase. Furthermore, Bagan [7] proved that, for a monadic second order query over classes of structures with bounded treewidth, the jth tuple in the query result with respect to an order $<$ can be computed in time $\log \|D\| + j$ after a linear time preprocessing phase.

In Chapter 9 we study unions of conjunctive queries.

We receive Theorem 3.10 for testing and Theorem 3.11 for enumerating.

Theorem 3.10 ([21])**.** *There is a dynamic algorithm that receives a t-hierarchical k-ary UCQ Q and a σ-db D_0, and computes within $t_p = poly(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows to test for an input tuple $\overline{a} \in \mathbf{dom}^k k$ if $\overline{a} \in Q(D)$ within time $t_t = poly(Q)$. Furthermore, the algorithm allows to answer a t-hierarchical Boolean UCQ within time $t_{ans} = O(1)$.*

Theorem 3.11 ([21])**.** *There is a dynamic algorithm that receives a q-hierarchical k-ary UCQ Q and a σ-db D_0, and computes within $t_p = poly(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows to enumerate $Q(D)$ with delay $t_d = poly(Q)$.*

A proof for Theorem 3.10 and Theorem 3.11 is given in Section 9.1. The results of these theorems has already been published in [21]. The proof of Theorem 3.10 is closely based on the proof in [21]. For Theorem 3.11 an alternative proof is given in Section 9.1. In Section 9.2 we look at the jth problem for unions of conjunctive queries. We identify a class of unions of conjunctive queries for which we obtain a result, the strongly exhaustively q-hierarchial UCQs. A UCQ $Q_1 \cup \cdots \cup Q_m$ is strongly exhaustively q-hierarchial if $\bigcap_{i \in [m]} Q_i$ is q-hierarchical. The corresponding theorem is the following.

Theorem 3.12. *There is a dynamic algorithm that receives a strongly exhaustively q-hierarchial k-ary UCQ Q and a σ-db D_0 of size $\|D_0\|$, and computes within preprocessing time $\exp(Q) \cdot O(\|D_0\| \log(\|D_0\|))$ a data structure that can be updated in time $\exp(Q) \cdot O(\log(\|D\|))$ and allows the following:*

(a) enumerate the tuples in $Q(D)$ with delay poly(Q) and

(b) upon input of an arbitrary number j, take time $\exp(Q) \cdot O(\log(\|D\|))$ to immediately output the jth tuple that the enumeration procedure of (a) would output,

where D is the current database.

It turns out that if a conjunctive query is not q-hierarchical, we cannot maintain the query result under updates. This thesis mainly considers the upper bound, the lower bounds can be found in [16]. These bounds are conditioned on the OMv-conjecture, a conjecture on the hardness of online matrix-vector multiplication that characterises the hardness of dynamic problems [57]. In [16], we obtain the following dichotomies, which are stated from the perspective of data complexity (i.e., the query is regarded to be fixed) and hold for any fixed $\varepsilon > 0$ under the OMv-conjecture. By n we denote in the remainder of this section the size of the active domain of the current database D. For the enumeration problem we restrict our attention to self-join free CQs, where every relation occurs only once in the query.

Theorem 3.13 ([16])**.** *Fix a number $\varepsilon > 0$ and a self-join free conjunctive query Q. If Q is not q-hierarchical, there is no algorithm with arbitrary preprocessing time and $O(n^{1-\varepsilon})$ update time that enumerates $Q(D)$ with $O(n^{1-\varepsilon})$ delay, unless the OMv-conjecture fails.*

For *Boolean* CQs we obtain a lower bound for *all* queries, i.e., also for queries that are not self-join free. To state the result, we need the standard notion of a homomorphic core. A *homomorphism* from a CQ $Q(x_1, \ldots, x_k)$ to a CQ $Q(y_1, \ldots, y_k)$ is a mapping h from vars(Q) to vars(Q) such that $h(x_i) = y_i$ for all $i \in [k]$ and if $Ru_1 \cdots u_r$ is an atom of Q, then $R\,h(u_1) \cdots h(u_r)$ is an atom of Q. The *homomorphic core* (for short, *core*) of a conjunctive query Q is a minimal subquery Q of Q such that there is a homomorphism from Q to Q, but no homomorphism from Q to a proper subquery of Q. By Chandra and Merlin's homomorphism theorem, every CQ Q has a unique (up to isomorphism) core Q' and $Q'(D) = Q(D)$ for all databases D (cf., e.g., [2]). While self-join free queries are their own cores, the situation is different for general CQs. Consider, for example, the queries

$$\begin{aligned} Q_1 &:= \big\{() \,:\, \exists x\, \exists y\, \big(Exx \wedge Exy \wedge Eyy\big)\big\} \quad \text{and} \\ Q_2 &:= \big\{() \,:\, \exists x\, \big(Exx\big)\big\}. \end{aligned}$$

Here, Q_2 is a core of Q_1 and thus $Q_1(D) = Q_2(D)$ for every database D. However, Q_2 is q-hierarchical, whereas Q_1 is not. The next lower bound theorem states that the result of a Boolean conjunctive query cannot be maintained efficiently if the query's core is not q-hierarchical.

Theorem 3.14 ([16]). *Fix a number $\varepsilon > 0$ and a Boolean conjunctive query Q. If the homomorphic core of Q is not q-hierarchical, then there is no algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that answers $Q(D)$ in time $t_{ans} = O(n^{2-\varepsilon})$, unless the OMv-conjecture fails.*

Let us now turn to the problem of computing the cardinality $|Q(D)|$ of the result of a query $Q = \{(x_1, \ldots, x_k) : \varphi\}$. From the Theorems 3.3 and 3.14 we know that we can efficiently decide whether $|Q(D)| > 0$ if and only if the homomorphic core of $\{() : \exists x_1 \cdots \exists x_k \, \varphi\}$ is q-hierarchical. The complexity of actually counting the number of tuples in $Q(D)$, however, depends on whether the core of the query $Q(x_1, \ldots, x_k)$ itself (rather than the core of its Boolean version $\{() : \exists x_1 \cdots \exists x_k \, \varphi\}$) is q-hierarchical. As in the Boolean case, the next theorem (together with Theorem 3.3) implies a dichotomy for all conjunctive queries. One difference is that we have to additionally rely on the OV-conjecture.

Theorem 3.15 ([16]). *Fix a number $\varepsilon > 0$ and a conjunctive query Q. If the homomorphic core of Q is not q-hierarchical, then there is no algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that computes $|Q(D)|$ in time $t_c = O(n^{1-\varepsilon})$, assuming the OMv-conjecture and the OV-conjecture.*

Last but not least, there is a new lower bound shown in this thesis for the *jth* problem, which can be shown by a reduction to the previous lower bound theorem:

Theorem 3.16. *Fix a number $\varepsilon > 0$ and a self-join free conjunctive query Q. If Q is not q-hierarchical, then there is no algorithm with arbitrary preprocessing time and $O(n^{1-\varepsilon})$ update time that solves the jth problem in time $O(n^{1-\varepsilon})$, unless the OMv-conjecture fails.*

A proof is given in Section 8.5.

4. Answering q-hierarchical conjunctive queries under updates

This chapter is devoted to the proof of Theorem 3.2 and Theorem 3.3, except for part (d) in Theorem 3.3. In Section 4.1 there is a definition of the data structure that will be used in both theorems. In Section 4.2 it is described how to initialise and update the data structure. In Section 4.3 is a proof for Theorem 3.2. Afterwards in Section 4.5, a lemma is presented that is very useful to prove the correctness of the enumeration algorithm and some facts for queries with aggregates. The proof of Theorem 3.3 will be continued with proving part (a) in Section 4.4, part (b) in Section 4.6, and part (c) in Section 4.7.

4.1. The fundamental data structure

We now give an alternative characterisation of q-hierarchical queries that sheds more light on their "tree-like" structure and will be useful for designing efficient query evaluation algorithms. Next, we define the notion of a *q-tree* for a CQ Q and show that Q is q-hierarchical if and only if it has a q-tree.

Definition 4.1. *Let Q be a CQ. A* q-tree *for Q is a rooted directed tree $T_Q = (V, E)$ with $V = \text{vars}(Q) \dot\cup \{v_{root}\}$ where*

(1) for all atoms ψ in Q the set $\text{vars}(\psi) \cup \{v_{root}\}$ forms a directed path in T_Q that starts from the root v_{root}, and

(2) if $\text{free}(Q) \neq \emptyset$, then $\text{free}(Q) \cup \{v_{root}\}$ is a connected subset in T_Q.

See Figure 4.1 for examples of q-trees. The following lemma gives a characterisation of the *q-hierarchical* conjunctive queries via q-trees.

Lemma 4.2. *A CQ Q is q-hierarchical if and only if Q has a q-tree. Moreover, there is a polynomial time algorithm that decides whether an input CQ Q is q-hierarchical, and if so, outputs a q-tree.*

Proof. The proof of the "only if" direction is easy, as every conjunctive query that has a q-tree T must be q-hierarchical, because if y is a descendant of x in T, then $\text{atoms}(y) \subseteq \text{atoms}(x)$ and if the path from v_{root} to x and the path from v_{root} to y are disjoint, then $\text{atoms}(y) \cap \text{atoms}(x) = \emptyset$.

For proving the "if" direction, we inductively construct a q-tree T_Q for all conjunctive queries Q with at most s variables. The induction basis for empty queries is

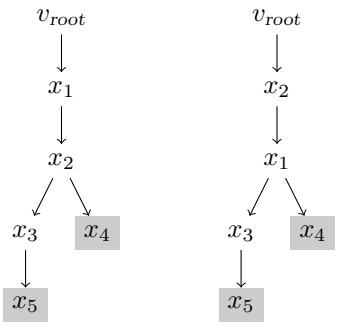


Figure 4.1.: Two q-trees for
$Q = \{(x_1, x_2, x_3) \ : \ \exists x_4 \exists x_5 \left(Ex_1x_2 \wedge Rx_4x_1x_2x_1 \wedge Rx_5x_3x_2x_1\right)\}$.

trivial. We just use the tree $T = (V, E)$ with $V = \{v_{root}\}$ and $E = \emptyset$. For the inductive step, we split the set of atoms in Q that have at least one variable, into disjoint sets $\Phi_1 \cup \cdots \cup \Phi_M$ we have

- for all $m \in [M]$ the following holds. For all $x, y \in \text{vars}(\Phi_m)$ we have $\text{atoms}(x) \subseteq \text{atoms}(y)$ or $\text{atoms}(x) \supseteq \text{atoms}(y)$ and
- for all $m, m' \in [M]$ with $m \neq m'$ the following holds. For all $x \in \text{vars}(\Phi_m)$ and $y \in \text{vars}(\Phi_{m'})$ we have that $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$.

Here, we define $\text{vars}(\Phi) := \bigcup_{\psi \in \Phi} \text{vars}(\psi)$ for every $\Phi \subseteq \text{atoms}(Q)$. It follows that for all $m, m' \in [M]$ with $m \neq m'$, we have that $\Phi_m \cap \Phi_{m'} = \emptyset$. To prove that fact let us assume for a contradiction that there is $m, m' \in [M]$ such that there is an atom $\psi \in \Phi_m \cap \Phi_{m'}$. Let $x \in \text{vars}(\psi)$ be arbitrary. Then, there is a $x \in \text{vars}(\Phi_m) \cap \text{vars}(\Phi_{m'})$ such that $\psi \in \text{atoms}(x)$. This contradicts to the second condition of the definition of Φ_m and $\Phi_{m'}$.

First of all, we show that $\Phi_m = \bigcup_{w \in \text{vars}(\Phi_m)} \text{atoms}(w)$. The fact $\Phi_m \subseteq \bigcup_{w \in \text{vars}(\Phi_m)} \text{atoms}(w)$ is clear. To show that $\Phi_m \supseteq \bigcup_{w \in \text{vars}(\Phi_m)} \text{atoms}(w)$, let us assume for a contradiction that there is a $\psi \in \bigcup_{w \in \text{vars}(\Phi_m)} \text{atoms}(w)$ with $\psi \notin \Phi_m$. Then, $\psi \in \Phi_{m'}$ for a $m' \neq m$. Since $\psi \in \text{atoms}(w)$ it follows that $w \in \text{vars}(\Phi_{m'})$. Then, there is a $w \in \text{vars}(\Phi_m) \cap \text{vars}(\Phi_{m'})$ with $\psi \in \text{atoms}(w)$. This violates the second condition in the definition of the sets $\Phi_1, \ldots, \Phi_M$.

Now, we show that for every $m \in [M]$ there is a $v \in \text{vars}(\Phi_m)$ such that v occurs in every atom in Φ_m. Let $v \in \text{vars}(\Phi_m)$ be a variable such that $|\text{atoms}(v)| = \max_{w \in \text{vars}(\Phi_m)} |\text{atoms}(w)|$. Thus, for all $w \in \text{vars}(\Phi_m) \backslash \{v\}$ it follows that $|\text{atoms}(w)| \leqslant |\text{atoms}(v)|$ and therefore $\text{atoms}(w) \not\supset \text{atoms}(v)$ and thus $\text{atoms}(w) \subseteq \text{atoms}(v)$. Because of this fact it follows that $\Phi_m = \bigcup_{w \in \text{vars}(\Phi_m)} \text{atoms}(w) = \text{atoms}(v)$ and therefore, v appears in every atom. We show now that if there are free variables in Φ_m, the variable v that appears in every atom in Φ_m is a free variable in Q, i.e., if $\text{free}(Q) \cap \text{vars}(\Phi_m) \neq \emptyset$, there must be a $v \in \text{free}(Q) \cap \text{vars}(\Phi_m)$ with $\Phi_m = \text{atoms}(v)$. For a contraction let us assume that for all $v \in \text{vars}(\Phi_m)$ with $\Phi_m = \text{atoms}(v)$ it holds

that $v \notin \text{free}(Q)$. Then, by the second condition of Definition 3.1 it follows that for all $w \in \text{vars}(\Phi_m)$ with $\Phi_m \neq \text{atoms}(w)$ that $w \notin \text{free}(Q)$ (since $\text{atoms}(w) \subset \text{atoms}(v)$) and therefore $\text{free}(Q) = \emptyset$, which violates the assumption.

If $M = 1$, let v be the variable that appears in every atom. Let Q' be the query we obtain if we delete every atom of the form Rv and delete every appearance of v in every atom. Note that by definition Q' is q-hierarchical with $s-1$ variables and let T' be the querytree of Q', we obtain from the induction hypothesis. Let T be the tree we obtain from T' if we replace the v_{root} node in T' by v and insert a "new root" v_{root} and an edge from v_{root} to v. Then, T is a q-tree for Q since for every atom $\psi \in \text{atoms}(Q')$ the set $(\text{vars}(\psi) \setminus \{v\}) \cup \{v_{root}\}$ forms a path in T' and by construction of T we have that the set $(\text{vars}(\psi) \setminus \{v\}) \cup \{v\} \cup \{v_{root}\}$ forms a path in T for every $\psi \in \text{atoms}(Q)$. If $\text{free}(Q) \neq \emptyset$, then $v \in \text{free}(Q)$ and therefore by the induction hypothesis it follows that $\text{free}(Q) \cup \{v_{root}\}$ is a connected subset in T.

If $M > 1$, for every $m \in [M]$ let Q_m be the query we obtain, by removing every atom in Q that does not belong to Φ_m and the variables that do not belong to $\text{vars}(\Phi_m)$. Note that $|\text{vars}(Q_m)| = |\text{vars}(\Phi_m)| < s$. Thus, let T_m be the q-tree we obtain from the induction hypothesis. For every $m \in [M]$ there must be a node v_m in T_m such that this is the only node that has an edge from the root node v_{root}. Since otherwise, if there are two nodes v_m^1 and v_m^2 with $v_m^1 \neq v_m^2$ and $\{v_{root}, v_m^1\}, \{v_{root}, v_m^2\} \in E(T_m)$, there is an atom $\psi_1 \in \text{atoms}(v_m^1)$ and $\psi_2 \in \text{atoms}(v_m^2)$, but $\text{vars}(\psi_1) \cap \text{vars}(\psi_2) = \emptyset$, which is a contradiction to the fact that there must be a variable that appears in every atom. Let T be the tree that we obtain from the q-trees $T_1, \ldots, T_M$ if we remove the v_{root} nodes in every tree and insert a new root node v_{root} and for all $m \in [M]$ the edge $\{v_{root}, v_m\}$, i.e., $T = (V, E)$ is the q-tree with $V := \bigcup_{m=1}^{M} V_m$ and $E := \bigcup_{m=1}^{M} E_m$ where $(V_m, E_m) = T_m$ for all $m \in [M]$. T is a q-tree for Q since for every atom $\psi \in \text{atoms}(Q)$ there is a $m \in [M]$ such that $\psi \in \Phi_m$ and $\text{vars}(\psi) \cup \{v_{root}\}$ forms a path in T_m. By construction, the path still appears in T. Furthermore, by the induction hypothesis it follows that $\text{free}(Q) \cup \{v_{root}\}$ is a connected subset of in T. Thus, T is a q-tree for Q.

It is easy to see that this construction can be computed in polynomial time. □

For the remainder of this chapter we assume that $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is a q-hierarchical conjunctive query, $\text{vars}(Q) = \{x_1, \ldots, x_m\}$ with $0 \leqslant k \leqslant m$, and Q is of the form

$$Q \quad = \quad \left\{(x_1 \ldots x_k, b_1, \cdots, b_\ell) \ : \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right)\right\}, \tag{4.1}$$

where $b_1, \ldots, b_\ell \in \mathbf{dom}$ and $\psi_1, \ldots, \psi_d$ are atomic queries of schema σ. We know that Q has a q-tree (see Lemma 4.2). We use the lemma's algorithm to construct in time $poly(Q)$ a q-tree T_Q of Q. For the remainder of this section, we simply write T to denote T_Q. Recall that the vertex set V of T is the set of variables in Q and v_{root}, i.e., $V = \{x_1, \ldots, x_m, v_{root}\}$.

The following notation will be convenient for describing and analysing our algorithms. For a node v of T, we write $\text{path}[v]$ to denote the set of all nodes of T that occur in the path from v_{root} to v in T (including v), and we let $\text{path}[v) := \text{path}[v] \setminus \{v\}$. Furthermore we let $\text{vpath}[v] := \text{path}[v] \setminus \{v_{root}\}$ and $\text{vpath}[v) := \text{path}[v) \setminus \{v_{root}\}$.

$child(v) := \{u : (v,u) \in E(T)\}$ is the set of children of v in T. The set $\mathsf{succ}(v)$, the successors of v, is the set of variables w in T (including v) such that there is a path from v to w in T.

A node v of T *represents* an atomic query ψ if and only if $\text{vpath}[v] = \text{vars}(\psi)$, i.e., the variables in $\text{vpath}[v]$ are exactly the variables in ψ.

An *assignment* is a partial mapping from **var** to **dom**. As usual, we write $\text{dom}(\alpha)$ for the domain of α. For a set $S \subseteq \mathbf{var}$, by $\alpha|_S$ we denote the restriction of α to $\text{dom}(\alpha) \cap S$. For $x \in \mathbf{var}$ and $a \in \mathbf{dom}$ we write $\alpha\frac{a}{x}$ for the assignment α' with domain $\text{dom}(\alpha) \cup \{x\}$, where $\alpha'(x) = a$ and $\alpha'(y) = \alpha(y)$ for all $y \in \text{dom}(\alpha) \setminus \{x\}$. An assignment β is called an *expansion* of α (for short: $\beta \supseteq \alpha$) if $\text{dom}(\beta) \supseteq \text{dom}(\alpha)$ and $\beta|_{\text{dom}(\alpha)} = \alpha$. The *empty assignment* $\emptyset$ is the assignment with empty domain. For pairwise distinct variables $v_1, \ldots, v_s$ and constants $a_1, \ldots, a_s \in \mathbf{dom}$ we write $\frac{a_1,\ldots,a_s}{v_1,\ldots,v_s}$ to denote the assignment α with $\text{dom}(\alpha) = \{v_1, \ldots, v_s\}$ and $\alpha(v_i) = a_i$ for all $i \in [d]$. With γ_{id} we denote the function with $\text{dom}(\gamma_{\mathsf{id}}) = \mathbf{dom}$ and $\gamma_{\mathsf{id}}(a) = a$ for all $a \in \mathbf{dom}$. For every conjunctive formula φ and for every assignment α with $\text{dom}(\alpha) = \text{vars}(\varphi)$ we write $(D, \alpha) \models \varphi$ if and only if for the valuation $\beta = \alpha \cup \gamma_{\mathsf{id}}$ holds $(D, \beta) \models \{(y_1, \ldots, y_s) : \varphi\}$ where $\{y_1, \ldots, y_s\} = \text{free}(\varphi)$.

We now describe the data structure `D` that will be built by the **preprocess** routine and maintained while executing the **update** routine. Our data structure for a given database D represents so-called *items*. Each item i is determined by an assignment $\alpha^i : \text{vpath}[v^i] \to \mathbf{dom}$ for some $v^i \in V$. We call α^i the *assignment of item* i, the last vertex v^i on the path the *variable of item* i and its value $a^i := \alpha^i(v^i)$ the *constant of item* i. An item i is denoted by $\texttt{[}\alpha^i\texttt{]} = \texttt{[}\frac{a_1,\ldots,a_s}{v_1,\ldots,v_s}\texttt{]}$, where $v_{root}, v_1, \ldots, v_s = v^i$ is the path from v_{root} to v^i in T. For every item i and every child u of v^i in T there is a doubly linked list $\mathcal{L}^i_u$ (the *u-list* of i) that contains items of the form $\texttt{[}\alpha^i\frac{b}{u}\texttt{]}$ and we have one pointer $child^i_u$ that points from i to the first element in $\mathcal{L}^i_u$. Note that in every data structure there is a designated *start*-item i_{start}, denoted by $\texttt{[}\emptyset\texttt{]}$, with the assignment $\alpha^{i_{start}} = \emptyset$ and $v^{i_{start}} = v_{root}$.

It is important to note that not every item of the form $\texttt{[}\alpha^i\frac{b}{u}\texttt{]}$ that is present in our data structure will be contained in the corresponding list $\mathcal{L}^i_u$. The *parent item* of an item $i = \texttt{[}\frac{a_1,\ldots,a_s}{v_1,\ldots,v_s}\texttt{]}$ is defined to be the item $\texttt{[}\frac{a_1,\ldots,a_{s-1}}{v_1,\ldots,v_{s-1}}\texttt{]}$. The *start* item has no parent item. As a convention, we set $\text{atoms}(v_{root}) = \text{atoms}(Q)$.

Let us now state which items are actually contained in our data structure. An item i is present in our data structure if and only if there is an atom $\psi \in \text{atoms}(v^i)$ such that there is an expansion $\beta \supseteq \alpha^i$ with $\text{dom}(\beta) = \text{vars}(\psi)$ and $(D, \beta) \models \psi$. It follows that every fact $R(a_1, \ldots, a_r)$ in the database gives rise to a constant number of items and that the overall number of items in our data structure is therefore linear in the size of the database. The definition also ensures that whenever an item is present in our data structure, then so is its parent item.

Let us now specify which of the present items are actually contained in the corresponding list $\mathcal{L}^i_u$.

Definition 4.3 (fit items)**.** *An item i is* fit *if and only if there is an expansion $\beta \supseteq \alpha^i$ such that $(D, \beta) \models \bigwedge_{\psi \in \text{atoms}(v^i)} \psi$.*

The doubly linked list $\mathcal{L}_u^i$ contains precisely those items that are fit. Being fit is a necessary requirement for participating in the query result and this is why we exclude unfit items from the lists. Note that whenever a tuple is inserted into or deleted from the database, this affects the "fit"-status of only a constant number of items. Furthermore, provided we have constant-time access to the items, we can update their status and include or exclude them from the corresponding lists in constant time.

The height of a variable v in a q-tree T_Q (or in a subtree T' of T_Q) is defined as the number of edges of the longest path from v to a leaf in T_Q (or in T', resp.). The height of an item i is defined as the height of its variable in T_Q (or in T', resp.). This will be convenient to prove facts via induction.

We say that a data structure is lexicographically ordered if for all present items i we have that for all $u \in \mathit{child}(v^i)$ the following holds. The items in the u-list of i are lexicographically ordered. This is a necessary requirement to enumerate the tuples in the query result in lexicographical order.

We consider now an example:

Example 4.4. *Let us consider the query*

$$Q = \{(y, x_1, x_2, x_3) \ : \ Eyx_1 \ \wedge \ Fyx_2x_3 \ \wedge \ Gyx_2x_3\}$$

and the database D_0 with $\mathbf{dom} = \mathbb{N}$ *and*

$$\begin{aligned} E^{D_0} \ &:= \ \big\{(1,1),\, (1,2),\, (1,3),\, (2,4),\, (2,8),\, (2,9),\, (3,2)\big\}, \\ F^{D_0} \ &:= \ \big\{(1,4,1),\, (1,5,2),\, (1,6,3),\, (1,6,4),\, (2,2,1),\, (2,2,8),\, (2,2,4), \\ &\qquad (3,1,1),\, (4,5,6)\big\}, \\ G^{D_0} \ &:= \ F^{D_0}. \end{aligned}$$

See Figure 4.2 for a q-tree of Q and Figure 4.3 for an illustration of the data structure of Q on D. An arrow from an item i to an item i' labelled by u denotes that i' is the first item of the u-list of i. A line between two items i (on the left) and i' (on the right) denotes that i' is the successor item of i in the doubly-linked list. The item $[\![\frac{4}{y}]\!]$ is present in the data structure but not fit since $(4,5,6) \in F^{D_0}$ and there is no $a \in \mathrm{adom}(D)$ with $(4,a) \in E^{D_0}$. All the other present items are fit.

Figure 4.2.: q-tree of T_Q from Example 4.4.

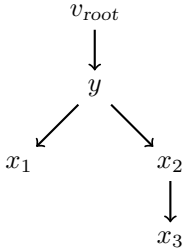
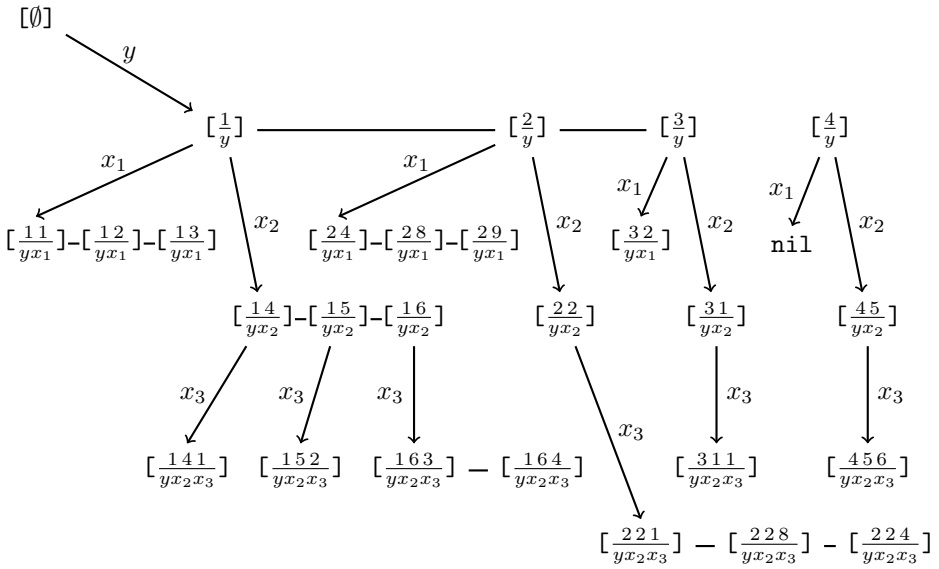


Figure 4.3.: Illustration of the data structure for the query and database in Example 4.4.



4.2. Preprocessing and updating the data structure

In the initialisation we compute the q-tree (by Lemma 4.2 this can be done in time $poly(Q)$), and we initialise the data structure for the empty database, i.e.,we initialise a *start*-item. In the preprocessing phase, we perform then initialisation and afterwards, $|D_0|$ insertion steps to ensure that the data structure represents the initial database D_0. By ensuring that the insertion time is $poly(Q)$, it follows that the preprocessing time is $poly(Q)O(\|D_0\|)$ and hence linear in the size of the initial database.

To illustrate how to update the data structure, consider again our database D_0 from Example 4.4 and suppose that the tuple $(4, 1)$ is inserted into the relation E. Now, we create the new item $[\frac{41}{yx_1}]$ that represents the new tuple and add it to the x_1-list of $[\frac{4}{y}]$. The item $[\frac{4}{y}]$ gets fit and thus, we add the item to the y-list of $[\emptyset]$. See Figure 4.4 for an illustration of the resulting database D_1.

Let us now suppose that the tuple $(3, 1, 1)$ will be deleted from G^{D_1}. As a consequence the items $[\frac{311}{yx_2x_3}]$, $[\frac{31}{yx_2}]$ and $[\frac{3}{y}]$ are not fit anymore but still present since $(3, 1, 1) \in F^{D_1}$. Therefore, we remove $[\frac{311}{yx_2x_3}]$ from the x_3-list of $[\frac{31}{yx_2}]$ and $[\frac{31}{yx_2}]$ from the x_2-list of $[\frac{3}{y}]$ and $[\frac{3}{y}]$ from the y-list of $[\emptyset]$. See Figure 4.5 for an illustration of the resulting database D_2.

As a last step in our example we also delete $(3, 1, 1)$ from F. Then the items $[\frac{311}{yx_2x_3}]$ and $[\frac{31}{yx_2}]$ are not allowed to be present in our data structure and therefore we delete them.

Figure 4.4.: Illustration of the data structure for the query and database in Example 4.4 after $(4, 1)$ was inserted to E^D.

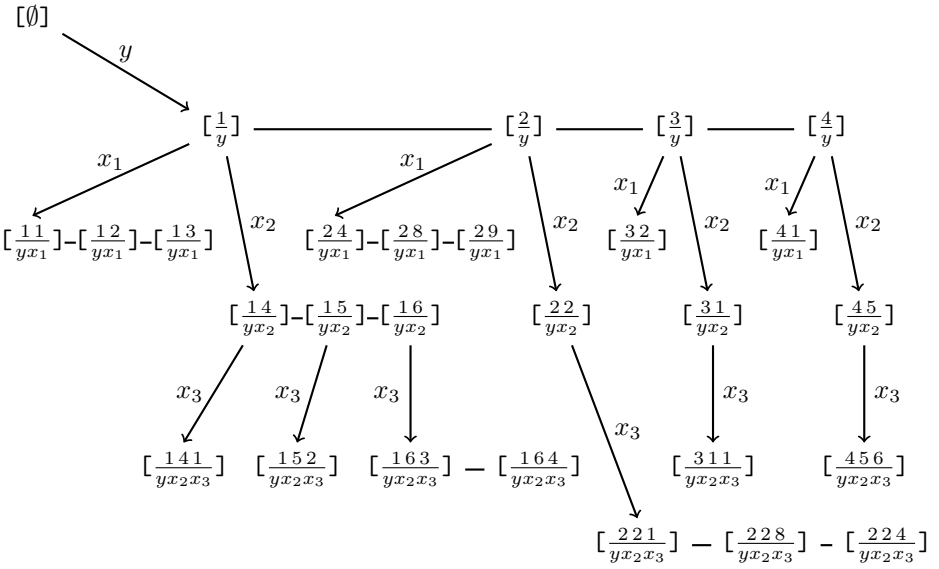
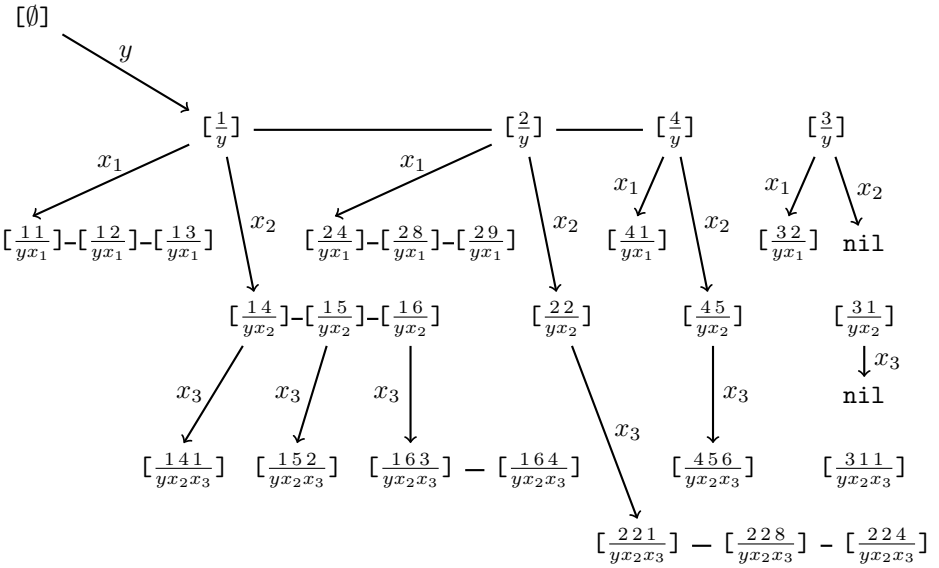


Figure 4.5.: Illustration of the data structure for the query and database in Example 4.4 after $(4, 1)$ was inserted to E^D and $(3, 1, 1)$ deleted from G^D.



When an update command arrives, we have to modify our data structure accordingly so that it meets the requirements described above. For convenience, we summarise the conditions below.

Condition 4.5. *(a) An item i is present in our data structure if and only if there is an atom $\psi \in \text{atoms}(v^i)$ such that there is an expansion $\beta \supseteq \alpha^i$ with $\text{dom}(\beta) = \text{vars}(\psi)$ and $(D, \beta) \models \psi$.*

(b) For every item i and every $u \in child(v^i)$ the list $\mathcal{L}^i_u$ consists of all fit items of the form $[\![\alpha^i \frac{b}{u}]\!]$.

The update procedure that guarantees that Condition 4.5 holds is given in Algorithm 1.

We also store for every $v \in \mathbf{var}(Q)$ a dictionary $\mathcal{D}_v$. Let $v_{root}, u_1, \ldots, u_s = v$ be the path from the root to v in the q-tree of Q. As keys we use $s-1$-ary tuples and the values are lists. In theses dictionaries we represent information that an item is present, i.e.,

$$a_s \text{ is in } \mathcal{D}_{u_s}[a_1, \ldots, a_{s-1}] \iff [\![\frac{a_1, \ldots, a_s}{u_1, \ldots, u_s}]\!] \text{ is present in the data structure}$$

We additionally store for every $R \in \sigma$ a dictionary $\mathcal{D}_R$ with $\text{ar}(R)$-ary keys and $\mathcal{D}_R[a_1, \ldots, a_r] = 1$ if and only if $(a_1, \ldots, a_r) \in R^D$ ($\mathcal{D}_R[a_1, \ldots, a_r] = 0$ otherwise). Furthermore we have to store a Boolean variable `start-is-fit` that is set to `True` if, and only if, the *start*-item $[\![\emptyset]\!]$ is fit. We will later use (for example in the enumeration algorithm) this variable to check in constant time if the *start*-item is fit. Algorithm 1 shows how to change the data structure after the databases is updated, i.e., a tuple is inserted or deleted.

The correctness of Algorithm 1 follows from the following claim:

Claim 4.6. *(a) Condition 4.5 holds for the data structure for the empty database.*

(b) If Condition 4.5 holds for the data structure, the condition still holds after an update was performed.

Proof. (a) For the empty database we do not have items except the *start*-item and therefore the condition holds.

(b) Let us assume that we have a database D and a data structure for D and Q for that Condition 4.5 holds and we receive an update and use Algorithm 1 to update the data structure. Let D' be the resulting database after the update was performed, i.e., $R^{D'} = R^D \cup \{(a_1, \ldots, a_r)\}$ and $S^{D'} = S^D$ for all $S \in \sigma \setminus \{R\}$ if we insert $(a_1, \ldots, a_r)$ to R and $R^{D'} = R^D \setminus \{(a_1, \ldots, a_r)\}$ and $S^{D'} = S^D$ for all $S \in \sigma \setminus \{R\}$ if we delete $(a_1, \ldots, a_r)$ from R.

Let us first consider the case that the if conditions in 8 and 10 do not hold.

The only items i that we consider in the algorithm are the following: There is an atom of the form $Rv_1, \ldots, v_r$ in Q and a $j \in [s]$ such that $\alpha^i = \frac{c_1, \ldots, c_j}{u_1, \ldots, u_j}$. Here

Algorithm 1 Procedure that modifies the data structure after an insert or delete command receives.

procedure UPDATE R($a_1, \ldots, a_r$)
  Set $\mathcal{D}_R[a_1, \ldots, a_r] = 1$ if UPDATE == insert.
  Set $\mathcal{D}_R[a_1, \ldots, a_r] = 0$ if UPDATE == delete.
  for all atoms of the form $Rv_1 \ldots v_r$ in Q **do**
    Let $u_1, \ldots, u_s$ be the nodes in a path in the q-tree T_Q such that $\{u_1, \ldots, u_s\} = \{v_1, \ldots, v_r\} \cap$ **var**.
    Let $c_1, \ldots, c_s$ be elements such that $c_j = a_m$ if and only if $u_j = v_m$.
    ▷ The next if-condition checks if the tuple *matches* to constants
    if there is an $i \in [r]$ with $v_i \in$ **dom** and $v_i \neq a_i$ **then**
      continue ▷ go to the next iteration step in the for loop
    if there are $i, j \in [r]$ with $v_i = v_j$ and $u_i \neq u_j$ **then**
      continue ▷ go to the next iteration step in the for loop
    if UPDATE == insert **then**
      for $j = 1$ **to** s **do**
        if $\left[\frac{c_1,\ldots,c_j}{u_1,\ldots,u_j}\right]$ does not exists in the data structure **then**
          Create $\left[\frac{c_1,\ldots,c_j}{u_1,\ldots,u_j}\right]$ and add c_j to $\mathcal{D}_{u_j}[c_1, \ldots, c_{j-1}]$.
      for $j = s$ **to** 1 **do**
        If $\left[\frac{c_1,\ldots,c_j}{u_1,\ldots,u_j}\right]$ becomes fit, add the item to x_j-list of $\left[\frac{c_1,\ldots,c_{j-1}}{u_1,\ldots,u_{j-1}}\right]$.
    if UPDATE == delete **then**
      for $j = s$ **to** 1 **do**
        if $\left[\frac{c_1,\ldots,c_j}{u_1,\ldots,u_j}\right]$ is not fit **then**
          Remove the item from the u_j-list of $\left[\frac{c_1,\ldots,c_{j-1}}{u_1,\ldots,u_{j-1}}\right]$.
        if Condition 4.5(a) does not hold for $\left[\frac{c_1,\ldots,c_j}{u_1,\ldots,u_j}\right]$ **then**
          Remove $\left[\frac{c_1,\ldots,c_j}{u_1,\ldots,u_j}\right]$ and remove c_j from $\mathcal{D}_{u_j}[c_1, \ldots, c_{j-1}]$.
    Set `start-is-fit` to `True` if and only if $[\emptyset]$ is fit.

and in the rest of the proof, $c_1, \dots, c_s, u_1, \dots, u_s$ are defined as in lines 5 and 6 in Algorithm 1. The corresponding atom in the iteration of the for-loop should be clear from context.

If we *insert* a tuple to a relation, we create all these items $[\frac{a_1,\dots,a_j}{u_1,\dots,u_j}]$, since $\alpha^i \subseteq \{v_\ell \mapsto a_\ell : \ell \in [r]\} =: \beta$ and it holds that $(D', \beta) \models \psi$ for an atom $\psi \in \text{atoms}(v^i)$, so they have to be present. We also check if they are fit and add them to the corresponding lists. If we *delete* a tuple from a relation, the algorithms checks for these items if they are still fit after the update and removes them from the corresponding lists if not and removes them from the data structure if they are not allowed to be present. Note that by definition of being fit or present such items can only receive the status of being present or fit if we insert a tuple, since we insert a tuple to D and lose their status of being present or fit after we delete a tuple from a relation.

Note that, for the other items i, that we do not consider in the algorithm, it holds that for all atoms of the form $Rv_1, \dots, v_r$ there is at least one $\{x \mapsto b\} \in \alpha^i$ such that $x \notin \{v_1, \dots, v_r\}$ or $b \notin \{a_1, \dots, a_r\}$. Furthermore, for these items i it holds that for every atom $\psi \in \text{atoms}(v^i)$ and for every expansion $\beta \supseteq \alpha^i$ with $\text{dom}(\beta) = \text{vars}(\psi)$ holds $(D, \beta) \models \psi \iff (D', \beta) \models \psi$ (and, in particular, $(D, \beta) \models \bigwedge_{\psi \in \text{atoms}(v^i)} \psi \iff (D', \beta) \models \bigwedge_{\psi \in \text{atoms}(v^i)} \psi$), since the update does not change relevant tuples for their satisfaction in the database. Thus, we obtain that these items do not change the status of being present and fit.

Let us now consider the case that at least one of the if conditions in Line 8 and 10 hold. Then, for the assignment β with $\beta(u_j) = c_j$ for all $j \in [s]$ it does not hold that $(D, \beta) \models Rv_1 \dots v_r$ and therefore such an update in consideration of the atomic formula $Rv_1 \dots v_r$ has no affect of the present status of any item in the data structure.

Therefore, Algorithm 1 guarantees that Condition 4.5 holds after an update was performed. □

In the remainder of this section we show that the algorithm takes time *poly*(Q). For all $v \in V(T)$ let $\text{exatoms}(v) := \left(\text{atoms}(v) \setminus \bigcup_{u \in child(v)} \text{atoms}(u)\right)$ if $v \neq v_{root}$ and $\text{exatoms}(v_{root})$ is the set of atoms with arity 0 in Q. First of all, we show how to quickly check that if an item is fit or not and whether an item is allowed to be present.

To do this, we use the following lemmas:

Lemma 4.7. *Let Q be a q-hierarchical CQ and D be a database. For all items i in the data structure for Q on D the following holds. The item i is fit if and only if for every $u \in child(v^i)$ is the u-list of i not empty and $(D, \alpha^i) \models \bigwedge_{\psi \in \text{exatoms}(v^i)} \psi$.*

The proof can be found at the end of this section (see page 39). To check if an item i is fit, we simply use the fact of Lemma 4.7 and check if there is a $u \in child(v^i)$ such that the u-list of i is empty and $(D, \alpha^i) \models \bigwedge_{\psi \in \text{exatoms}(v^i)} \psi$. To verify if $(D, \alpha^i) \models \bigwedge_{\psi \in \text{exatoms}(v^i)} \psi$ we lookup for every $Rx_1, \dots, x_r \in \text{exatoms}(v^i)$ if $(\beta(x_1), \dots, \beta(x_r))$

is in $\mathcal{D}_R$ for $\beta := \alpha^i \cup \gamma_{\mathsf{id}}$. Note that checking if an item is fit takes $poly(Q)$ time, since we have at most $|Q|$ lookups and we check for $|child(v^i)| \leqslant poly(Q)$ lists, whether they are empty. Note that we update the items in the data structure bottom up, i.e., every time we check if an i is fit, the items whose assignment that are extensions of α^i, already have an updated status (the information if they are fit or present).

To check if an item has to be present, we use the following lemma.

Lemma 4.8. *Let Q be a q-hierarchical CQ and D be a database. For all items i in the data structure for Q on D it holds that i is present if and only if there is an atom $\psi \in \text{exatoms}(v^i)$ such that $(D, \alpha^i) \models \psi$ or there is a $u \in child(v^i)$ and a $b \in \text{adom}(D)$ such that there is a present item $\hat{\imath} = [\![\alpha^i \frac{b}{u}]\!]$.*

The proof is given at the end of this section (see page 40).

To check if we have to delete an item from our data structure in line 22 of Algorithm 1, we check if

1. there is an atom $\psi \in \text{exatoms}(v^i)$ such that $(D, \alpha^i) \models \psi$ or
2. there is a $u \in child(v^i)$ and a $b \in \text{adom}(D)$ such that there is a present item $\hat{\imath} = [\![\alpha^i \frac{b}{u}]\!]$.

To check if condition 1 holds we lookup for all $\psi = Rx_1 \dots x_r \in \text{exatoms}(v^i)$ if $(\beta(x_1), \dots, \beta(x_r))$ is in $\mathcal{D}_R$ where $\beta = \alpha^i \cup \gamma_{\mathsf{id}}$. For condition 2, we check if $\mathcal{D}_{x_r}[\alpha(x_1), \dots, \alpha(x_{r-1})]$ is not empty.

Since lines 8, 10, 17, 20, 22 and 24 in Algorithm 1 take time $poly(Q)$ and the for loop has at most $\|Q\|$ iterations and the other lines take constant time, we need time $poly(Q)$ for Algorithm 1. Thus, we can update the data structure, that is not lexicographically ordered, in time $poly(Q)$ if we receive an update.

Remark 4.9. *Note that for lexicographically ordered data structures we have to take into account that for all present item i in the data structure and for all $u \in child(v^i)$ in the data structure the u-list of i is lexicographically ordered. To realise this, we have to create for all lists ordered data structures such as AVL-trees or red-black trees (see [32] for a survey). For these data structures it takes time $O(\log(|\text{adom}(D)|))$ to insert or to delete an element from the list. By analysing the running time for the update algorithm with the assumption that it takes time $O(\log(|\text{adom}(D)|))$ to insert or delete an element from a u-list of i, we obtain that it takes time $\log(|\text{adom}(D)|)\, poly(Q)$ to update lexicographically ordered data structures.*

In the remainder of this section we give proofs for Lemma 4.7 and Lemma 4.8.

Proof of Lemma 4.7. Let us assume that a present item i is fit. Then, there is a $\beta \supseteq \alpha^i$ such that $(D, \beta) \models \bigwedge_{\psi \in \text{atoms}(v^i)} \psi$. In particular, it holds that $(D, \alpha^i) \models \bigwedge_{\psi \in \text{exatoms}(v^i)} \psi$ and for every $w \in child(v^i)$ it holds that $\text{atoms}(w) \subseteq \text{atoms}(v^i)$ and therefore $(D, \beta) \models \bigwedge_{\psi \in \text{atoms}(w)} \psi$. Thus, there is an assignment $\alpha_w = \beta|_{\text{vpath}[w]} \subseteq \beta$ such that $(D, \beta) \models \bigwedge_{\psi \in \text{atoms}(w)} \psi$ and, in particular, the item $[\![\alpha_w]\!]$ is present and fit. Therefore, there must be an item $[\![\alpha_w]\!]$ in the w-list of i for every $w \in child(v^i)$.

For the "only if" direction let us assume that every u-list of i is not empty and $(D, \alpha^i) \models \psi$ for all $\psi \in \text{exatoms}(v^i)$. Then, there exists for every $u \in \mathit{child}(v^i)$ an item $\lfloor \alpha^i \frac{b}{u} \rfloor$ that is fit, i.e., there is a $\beta_u \supseteq \alpha^i \frac{b}{u}$ such that $(D, \beta_u) \models \bigwedge_{\psi \in \text{atoms}(u)} \psi$. Let $\beta := \bigcup_{u \in \mathit{child}(v^i)} \beta_u$. Note that β is a valid assignment since $\alpha^i \subseteq \beta_u$ for all $u \in \mathit{child}(v^i)$ and $\text{dom}(\alpha^i) = \bigcap_{u \in \mathit{child}(v^i)} \text{dom}(\beta_u)$ and $\text{atoms}(v^i) = \bigcup_{u \in \mathit{child}(v^i)} \text{atoms}(v_i) \cup \text{exatoms}(v)$. It follows that $(D, \beta) \models \bigwedge_{\psi \in \text{atoms}(v^i)} \psi$ and i is therefore fit. □

Proof of Lemma 4.8. Let us assume that i is a present item in the data structure and let $\psi \in \text{atoms}(v^i)$ be the atom and $\beta \supseteq \alpha^i$ be the assignment with $\text{dom}(\beta) = \text{vars}(\psi)$ and $(D, \beta) \models \psi$.

- *Case 1:* $\psi \in \text{exatoms}(v^i)$. Then, $\text{dom}(\beta) = \text{vars}\psi = \text{dom}(\alpha^i)$ and thus, $(D, \alpha^i) \models \psi$.
- *Case 2:* $\psi \notin \text{exatoms}(v^i)$. Then, there exists a $u \in \mathit{child}(v^i)$ such that $\psi \in \text{atoms}(u)$. Let $b = \beta(u)$. Then, it holds that $\beta \supseteq \alpha^i \frac{b}{u}$ with $\text{dom}(\beta) = \text{vars}(\psi)$ and $(D, \beta) \models \psi$ and, in particular, there is a present item $\lfloor \alpha^i \frac{b}{u} \rfloor$ in the data structure.

For the "only if" direction, we consider two cases:

- *Case 1:* There is an atom $\psi \in \text{exatoms}(v^i)$ such that $(D, \alpha^i) \models \psi$. It is straightforward to verify that i is present in the data structure.
- *Case 2:* $\lfloor \alpha^i \frac{b}{u} \rfloor$ is present in our data structure. Then, there is an atom $\psi \in \text{atoms}(u) \subseteq \text{atoms}(v^i)$ and an expansion $\beta \supseteq \alpha^i \frac{b}{u} \supseteq \alpha^i$ with $\text{dom}(\beta) = \text{vars}(\psi)$ and $(D, \beta) \models \psi$. By definition, i is present in the data structure. □

4.3. Answering Boolean q-hierarchical queries

The aim of this section is to give a proof for Theorem 3.2, i.e., to show how to use the data structure for answering Boolean q-hierarchical conjunctive queries.

For convenience, Theorem 3.2 is restated in the following.

Theorem 3.2. *There is a dynamic algorithm that receives a Boolean q-hierarchical conjunctive query Q and a σ-db D_0, and computes within $t_i = \text{poly}(Q)$ initialisation time and $t_p = \text{poly}(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = \text{poly}(Q)$ and allows to answer $Q(D)$ in time $t_{ans} = O(1)$ where D is the current database.*

Proof of Theorem 3.2. Let

$$Q := \{(b_1, \ldots, b_\ell) : \exists y_1 \cdots \exists y_k \varphi\}$$

be the input Boolean q-hierarchical CQ and let D be the input database. The main idea is to use the data structure from this chapter to maintain the query Q' on D under updates where

$$Q' := \{(y_1, \ldots, y_k, b_1, \ldots, b_\ell) : \varphi\}$$

with the **init**, **preprocess** and the **update** routines from Section 4.2. An update can be done in time $t_u = \mathit{poly}(Q)$, the initialisation can be done in time $t_i = \mathit{poly}(Q)$ and the preprocessing can be done in time $t_p = \|D_0\| \mathit{poly}(Q)$.

For the **answer** routine, we just check if the variable `start-is-fit` is `true` and return the corresponding result. This can be done in time $O(1)$. The routine is correct, since by Definition 4.3 it follows that the start item is fit, if and only, if there is an assignment β such that $(D, \beta) \models \bigwedge_{\psi \in \mathrm{atoms}(v_{root})} \psi$. Thus, $(D, \beta) \models Q'$. Obviously, this fact holds if and only if $Q(D) = \texttt{yes}$.

This concludes the proof of Theorem 3.2. □

4.4. Testing q-hierarchical queries

The aim of this section is to show Theorem 3.3 (a), i.e., to show how the test routine for q-hierarchical queries works.

First we consider an example.

Example 4.10. *Let us recall the database D_0 and the query Q from Example 4.4. Assume, that we want to test if $(1,3,6,3)$ belongs to the result set. Let $\alpha := \frac{1\ 3\ 6\ 3}{yx_1x_2x_3}$ be the assignment such that it belongs to the tuple if matching the free variables with the components of the tuple, i.e., $(1,3,6,3) = (\alpha(y), \alpha(x_1), \alpha(x_2), \alpha(x_3))$.*

For all $u \in \mathrm{free}(Q)$ we consider the assignments $\alpha_u := \alpha|_{\mathrm{vpath}[u]}$ and check if the items $[\emptyset]$ and $[\alpha_u]$ are fit for all $u \in \mathrm{free}(Q)$. The following holds.

- *Since $[\emptyset]$ is fit, it follows that there are $a_1, a_2, a_3, a_4 \in \mathrm{adom}(D_0)$ such that $(a_1, a_2) \in E^{D_0}$ and $(a_1, a_3, a_4) \in F^{D_0} \cap G^{D_0}$.*
- *Since $[\frac{1}{y}]$ is fit, it follows that there are $a_2, a_3, a_4 \in \mathrm{adom}(D_0)$ such that $(1, a_2) \in E^{D_0}$ and $(1, a_3, a_4) \in F^{D_0} \cap G^{D_0}$.*
- *Since $[\frac{1\ 3}{yx_1}]$ is fit, it follows that $(1,3) \in E^{D_0}$.*
- *Since $[\frac{1\ 6}{yx_2}]$ is fit, it follows that there is a $a_4 \in \mathrm{adom}(D_0)$ such that $(1,6,a_4) \in F^{D_0} \cap G^{D_0}$.*
- *Since $[\frac{1\ 6\ 3}{yx_2x_3}]$ is fit, it follows that $(1,6,3) \in F^{D_0} \cap G^{D_0}$.*

Since $(1,3) \in E^{D_0}$ and $(1,6,3) \in F^{D_0} \cap G^{D_0}$, it follows that $(1,3,6,3)$ belongs to the query result.

Assume, that we want to test if $(1,4,6,3)$ belongs to the query result. There is no fit and present item $[\frac{1\ 4}{yx_1}]$ and thus $(1,4) \notin E^{D_0}$ and therefore, $(1,4,6,3)$ does not belong to the query result.

For the remainder of this section we assume that $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is a q-hierarchical conjunctive query, $\mathrm{vars}(Q) = \{x_1, \ldots, x_m\}$ with $0 \leqslant k \leqslant m$, and Q is of the form

$$Q \quad = \quad \left\{(x_1 \ldots x_k, b_1, \cdots, b_\ell) \ : \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right)\right\}, \qquad (4.2)$$

where $b_1, \ldots, b_\ell \in \mathbf{dom}$ and $\psi_1, \ldots, \psi_d$ are atomic queries of schema σ. We are given as input Q and a σ-db D and let us assume we have the data structure from Section 4.1 for Q and D. When we call the **test** routine with input $\overline{a} = (\alpha(x_1), \ldots, \alpha(x_k), c_1, \ldots, c_\ell)$, we check if the following two conditions are true.

- $b_j = c_j$ for all $j \in [\ell]$.
- For all $v \in \mathrm{vars}(Q)$ the items $\mathtt{[}\alpha|_{\mathrm{vpath}[v]}\mathtt{]}$ are fit and output the result of the test.

We output `Yes` if and only if the results of both tests are true.

Note that by Definition 4.3, it follows that

$$\begin{aligned}(D,\alpha) \models \psi \text{ for all atoms } \psi \in \mathrm{atoms}(Q)\\ \Leftrightarrow \quad (D,\alpha) \models \psi \text{ for all atoms } \psi \in \mathrm{atoms}(v) \text{ for all } v \in \mathrm{vars}(Q)\\ \text{and for all atoms } \psi \text{ with artiy } 0\\ \Leftrightarrow \quad \mathtt{[}\alpha|_{\mathrm{vpath}[v]}\mathtt{]} \text{ is fit for all } v \in \mathrm{vars}(Q)\end{aligned}$$

and therefore the **test** routine is correct. Checking the first condition can be done in $O(poly(Q))$. Using Lemma 4.7, we test for an item i if for every $u \in child(v^i)$ the u-list of i is not empty and if $(D, \alpha^i) \models \bigwedge_{\psi \in \mathrm{exatoms}(v^i)} \psi$. With the data structure in Section 4.1, this can be done in $O(poly(Q))$ time. This has to be done $|\mathrm{vars}(Q)|$ times and therefore the **test** routine takes time $O(poly(Q))$. This concludes the proof of Theorem 3.3 (a).

4.5. The decomposition lemma

This section presents a lemma which is a very convenient tool to obtain a lot of results and it will be used in various manners. It gives the conditions that holds for every present and fit item in the data structure for obtaining the result set from the data structure. Already in the next Section 4.6 there is an application given, how to use this lemma to obtain the result for enumeration.

Before we give the lemma, we need a definition.

Definition 4.11 (Extensions sets for items)**.** *For all present items i in the data structure let*

$$\mathcal{E}^i := \left\{ \beta \supseteq \alpha^i \;:\; \mathrm{dom}(\beta) = \bigcup_{\psi \in \mathrm{atoms}(v^i)} \mathrm{vars}(\psi),\ (D,\beta) \models \bigwedge_{\psi \in \mathrm{atoms}(v^i)} \psi \right\}.$$

Furthermore, we let

$$\widetilde{\mathcal{E}}^i := \left\{ \beta|_{\mathrm{free}(Q)} \,:\, \beta \in \mathcal{E}^i \right\}. \tag{4.3}$$

Recall that $\mathrm{atoms}(v_{root}) = \mathrm{atoms}(Q)$ and $v^{\mathtt{[}\emptyset\mathtt{]}} = v_{root}$. It follows that $Q(D) = \{(\alpha(x_1), \ldots, \alpha(x_k), b_1, \ldots, b_\ell) \,:\, \alpha \in \widetilde{\mathcal{E}}^{\mathtt{[}\emptyset\mathtt{]}}\}$ if $\mathtt{[}\emptyset\mathtt{]}$ is fit. The decomposition lemma for q-hierarchical queries shows us how the sets $\widetilde{\mathcal{E}}^i$ for all fit items i in the data structure can be decomposed:

Lemma 4.12 (Decomposition Lemma for q-hierarchical queries)**.** *Let Q be a q-hierarchical CQ and let D be a σ-db and let T_Q be a q-tree of Q. Let i be a present and fit item in the data structure of Q on D and let $u_1, \ldots, u_s$ be the children of v^i in T_Q. Then,*

$$\mathcal{E}^i \;=\; \left\{ \bigcup_{j=1}^{s} \alpha_j \cup \alpha^i : \text{ for all } j \in [s] \text{ there is an } \widehat{\iota} \in \mathcal{L}^i_{u_j} \text{ such that } \alpha_j \in \mathcal{E}^{\widehat{\iota}} \right\}$$

and

$$\widetilde{\mathcal{E}}^i \;=\; \left\{ \bigcup_{j=1}^{s} \alpha_j \cup \alpha^i : \begin{array}{l} \text{for all } j \in [s] \text{ with } u_j \in \mathrm{free}(Q) \text{ there is a } \widehat{\iota} \in \mathcal{L}^i_{u_j} \\ \text{such that } \alpha_j \in \widetilde{\mathcal{E}}^{\widehat{\iota}} \end{array} \right\}.$$

Proof. Let $\alpha \in \mathcal{E}^i$. Then by definition, $\alpha \supseteq \alpha^i$ is an extension with $\mathrm{dom}(\alpha) = \bigcup_{\psi \in \mathrm{atoms}(v^i)} \mathrm{vars}(\psi)$ such that, for all $\psi \in \mathrm{atoms}(v^i)$ we have that $(D, \alpha) \models \psi$. Note that by construction of the q-tree the set $\mathrm{atoms}(v^i)$ can be decomposed into disjoint sets $\mathrm{atoms}(v^i) = \mathrm{exatoms}(v^i) \,\dot\cup\, \mathrm{atoms}(u_1) \,\dot\cup\, \cdots \,\dot\cup\, \mathrm{atoms}(u_s)$, where $u_1, \ldots, u_s$ are the children of v^i in T_Q. In particular, we obtain that for all atoms $\psi \in \mathrm{exatoms}(v^i)$ we have $(D, \alpha) \models \psi$ and for all $j \in [s]$ it holds that for all $\psi \in \mathrm{atoms}(u_j)$ is $(D, \alpha) \models \psi$ since i is fit. Furthermore we can restrict the domain of the assignment α to variables, that appear in the atom and we obtain for all atoms $\psi \in \mathrm{exatoms}(v^i)$ that $(D, \alpha|_{\mathrm{vpath}[v^i]}) \models \psi$ and for all $j \in [s]$ we have that for all $\psi \in \mathrm{atoms}(u_j)$ it holds that $(D, \alpha|_{\mathrm{vpath}[v^i] \cup \mathsf{succ}(u_j)}) \models \psi$. By definition of fit, we obtain for all $j \in [k]$ that the item $\lceil \alpha|_{\mathrm{vpath}[u_j]} \rceil$ is fit (since for the assignment $\alpha|_{\mathrm{vpath}[v^i] \cup \mathsf{succ}(u_j)}$ it holds that $(D, \alpha|_{\mathrm{vpath}[v^i] \cup \mathsf{succ}(u_j)}) \models \psi$ for all $\psi \in \mathrm{atoms}(u_j)$) and thus $\hat{\iota} = \lceil \alpha|_{\mathrm{vpath}[u_j]} \rceil$ is included in the u_j-list of i. Furthermore, by definition, $\alpha|_{\mathrm{vpath}[v^i] \cup \mathsf{succ}(u_j)} \in \mathcal{E}^{\hat{\iota}}$. We have that

$$\alpha \in \left\{ \bigcup_{j=1}^{s} \alpha_j \cup \alpha^i : \text{ for all } j \in [s] \text{ there is a } \widehat{\iota} \in \mathcal{L}^i_{u_j} \text{ such that } \alpha_j \in \mathcal{E}^{\widehat{\iota}} \right\}.$$

It remains to show that every assignment on the right side of the equation belongs to $\mathcal{E}^i$. Let $\alpha = \alpha_1 \cup \cdots \cup \alpha_s$ be an assignment such that there is a $\widehat{\iota_j} \in \mathcal{L}^i_{u_j}$ where $\alpha_j \in \mathcal{E}^{\widehat{\iota_j}}$. By definition, for every $\psi \in \mathrm{atoms}(u_j)$ it holds that $(D, \alpha_j) \models \psi$ for all $j \in [s]$ and, in particular, $(D, \alpha) \models \psi$. Since i is fit, it follows that for all $\psi \in \mathrm{exatoms}(v^i)$ we have $(D, \alpha^i) \models \psi$ and, in particular, $(D, \alpha) \models \psi$. Since $\mathrm{atoms}(v^i) = \mathrm{exatoms}(v^i) \,\dot\cup\, \mathrm{atoms}(u_1) \,\dot\cup\, \cdots \,\dot\cup\, \mathrm{atoms}(u_s)$, it holds that $\alpha \in \mathcal{E}^i$.

For the second equation, we use for all items i with $v^i \in \mathrm{free}(Q)$ the following argument.

$$\alpha|_{\text{free}(Q)} \in \widetilde{\mathcal{E}}^i \iff \alpha \in \mathcal{E}^i$$

$$\iff \alpha \in \left\{ \bigcup_{j=1}^{s} \alpha_j \cup \alpha^i : \begin{array}{c}\text{for all } j \in [s] \text{ there is a} \\ \widehat{\iota} \in \mathcal{L}^i_{u_j} \text{ such that } \alpha_j \in \mathcal{E}^{\widehat{\iota}}\end{array} \right\}$$

$$\overset{(\star)}{\iff} \alpha|_{\text{free}(Q)} \in \left\{ \bigcup_{j=1}^{s} \alpha_j|_{\text{free}(Q)} \cup \alpha^i : \begin{array}{l}\text{for all } j \in [s] \text{ where } u_j \in \text{free}(Q) \\ \text{there is a } \widehat{\iota} \in \mathcal{L}^i_{u_j} \\ \text{such that } \alpha_j|_{\text{free}(Q)} \in \widetilde{\mathcal{E}}^{\widehat{\iota}} \\ \text{and } \alpha_j := \alpha^i \text{ for all } j \in [s] \\ \text{with } u_j \notin \text{free}(Q)\end{array} \right\}$$

The "$\Longrightarrow$"-direction of $(\star)$ is easy to see. To verify that the "$\Longleftarrow$"-direction holds let β be an assignment of the form $\beta = \beta_1 \cup \cdots \cup \beta_s \cup \alpha^i$ where for all $j \in [s]$ with $u_j \in \text{free}(Q)$ there is a $\widehat{\iota} \in \mathcal{L}^i_{u_j}$ such that $\beta_j \in \widetilde{\mathcal{E}}^{\widehat{\iota}}$ and for all $j \in [s]$ with $u_j \notin \text{free}(Q)$ we have $\beta_j = \alpha^i$. By the definition of $\widetilde{\mathcal{E}}^i$ and by the fact that i is fit, we have that for all $j \in [s]$ with $u_j \in \text{free}(Q)$ there is a $\widehat{\iota} \in \mathcal{L}^i_{u_j}$ and a $\alpha_j \in \mathcal{E}^{\widehat{\iota}}$ where $\alpha_j|_{\text{free}(Q)} = \beta_j$ and for all $j \in [s]$ with $u_j \notin \text{free}(Q)$ we have that there is a $\widehat{\iota} \in \mathcal{L}^i_{u_j}$ and a $\beta_j \in \mathcal{E}^{\widehat{\iota}}$ where $\alpha_j|_{\text{free}(Q)} = \alpha^i$. Let $\alpha := \alpha_1 \cup \cdots \cup \alpha_s \cup \alpha^i$. Clearly $\alpha|_{\text{free}(Q)} = \beta$ and

$$\alpha \in \left\{ \bigcup_{j=1}^{s} \alpha_j \cup \alpha^i : \begin{array}{c}\text{for all } j \in [s] \text{ there is a} \\ \widehat{\iota} \in \mathcal{L}^i_{u_j} \text{ such that } \alpha_j \in \mathcal{E}^{\widehat{\iota}}\end{array} \right\}.$$

□

4.6. Enumerating the query result with constant delay

The aim of this chapter is to prove Theorem 3.3(b), i.e. we now discuss how the data structure from the previous chapter can be used to enumerate the query result with constant delay. See Table 4.1 for the 22 result tuples of Example 4.4.

In the next example we show how to algorithmically enumerate the tuples in Table 4.1 using the data structure for D_0 and Q from Example 4.4

Example 4.13. *Let us consider the query Q and the database D_0 from Example 4.4. Algorithm 2 enumerates the tuples in the query result. The algorithm runs through the data structure and it runs over all fit items. Since all the items are fit and the output tuples comes from the assignments of fit item, it follows that the tuples are part of the query result.*

Until we reach the first output in our example, the item in line 2 is set to $[\frac{1}{y}]$, the item in line 3 is set to $[\frac{1\ 1}{y\ x_1}]$, the item in line 4 is set to $[\frac{1\ 4}{y\ x_2}]$ and the item in line 5 is set to $[\frac{1\ 4\ 1}{y\ x_2 x_3}]$. Since $[\frac{1\ 4\ 1}{y\ x_2 x_3}]$ is the item in the x_3-list of $[\frac{1\ 4}{y\ x_2}]$ and thus, in we

y	x_1	x_2	x_3
1	1	4	1
1	1	5	2
1	1	6	3
1	1	6	4
1	2	4	1
1	2	5	2
1	2	6	3
1	2	6	4
1	3	4	1
1	3	5	2
1	3	6	3
1	3	6	4
2	4	2	1
2	4	2	8
2	4	2	4
2	8	2	1
2	8	2	8
2	8	2	4
2	9	2	1
2	9	2	8
2	9	2	4
3	2	1	1

Table 4.1.: Enumeration of $Q(D_0)$ from Example 4.4.

Algorithm 2 Enumeration algorithm.

1: **if** $[\emptyset]$ is fit **then**
2: **for** $[\frac{a_0}{y}]$ in the y-list of $[\emptyset]$ **do**
3: **for** $[\frac{a_0 a_1}{y\, x_1}]$ in the x_1-list of $[\frac{a_0}{y}]$ **do**
4: **for** $[\frac{a_0 a_2}{y\, x_2}]$ in the x_2-list of $[\frac{a_0}{y}]$ **do**
5: **for** $[\frac{a_0 a_2 a_3}{y\, x_2 x_3}]$ in the x_3-list of $[\frac{a_0 a_2}{y\, x_2}]$ **do**
6: **Output** (a_0, a_1, a_2, a_3)
7: **Output** EOE

set the item in line 4 is set to $\texttt{[}\frac{1\ 5}{y\ x_2}\texttt{]}$ *and in line 5 is set to* $\texttt{[}\frac{1\ 5\ 2}{y\ x_2 x_3}\texttt{]}$*. The algorithm continues with enumerating all results.*

For the remainder of this chapter we assume that $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is a q-hierarchical conjunctive query, $\text{vars}(Q) = \{x_1, \ldots, x_m\}$ with $0 \leqslant k \leqslant m$, and Q is of the form

$$Q \quad = \quad \left\{(x_1 \ldots x_k, b_1, \cdots, b_\ell) \ : \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right)\right\}, \tag{4.4}$$

where $b_1, \ldots, b_\ell \in \mathbf{dom}$ and $\psi_1, \ldots, \psi_d$ are atomic queries of schema σ. To enumerate the result of a non-Boolean conjunctive q-hierarchical query $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$, let T_{free} be the subtree of T_Q induced on $V(T_{\text{free}}) := \text{free}(Q) \cup \{v_{root}\} = \{x_1, \ldots, x_k, v_{root}\}$. For each node v of T_{free}, let us fix an (arbitrary) linear order $\leqslant^v$ for all variables v in Q. In our example query we have $T_{\text{free}} = T$ and we let $x_1 <^y x_2$. For the enumeration procedure we will use the notion of $\mathcal{E}^i$ (see Definition 4.11) and the decomposition of these sets given in Lemma 4.12.

The main idea for the **enumerate** routine is the following. If the *start*-item $\texttt{[}\emptyset\texttt{]}$ is not fit, the **enumerate** routine stops immediately with output `EOE`. Otherwise, we do the following. Inductively, for every item i we enumerate the set $\widetilde{\mathcal{E}}^i$ using the following algorithm. We iterate for all $u \in \mathit{child}(v^i) \cap \text{free}(Q)$ over all $\widehat{\imath} \in \mathcal{L}_u^i$ over all assignments in $\alpha_u \in \widetilde{\mathcal{E}}^{\widehat{\imath}_j}$ and construct the assignment $\alpha^i \cup \bigcup_{u \in \mathit{child}(v^i)} \alpha_u$. By using Lemma 4.12 we know that we enumerate the assignments in $\widetilde{\mathcal{E}}^i$. If we enumerate $\widetilde{\mathcal{E}}^{\texttt{[}\emptyset\texttt{]}}$, we can easily construct the result tuples in $Q(D)$ by outputting $(\alpha(x_1), \ldots, \alpha(x_k), b_1, \ldots, b_\ell)$ for every $\alpha \in \widetilde{\mathcal{E}}^{\texttt{[}\emptyset\texttt{]}}$. The pseudo-code for the described recursive **enumerate** routine is given in Algorithm 3. The ENUM function in Algorithm 3 also requests a set of an order for every node. This order defines in which grouping the result set will be output, since the order can change the sorting of the for loops. For example, in Table 4.1 we used the order $x_1 <^y x_2$ for y. This implies by construction of the algorithm, that the result set is grouped by the variables y and x_1.

The next lemma establishes the correctness of Algorithm 3 and show that the tuples will be enumerated with delay $O(|\text{vars}(Q)|)$ without repetition.

Lemma 4.14. *Let Q be a q-hierarchical CQ and let D be a σ-db. For all present and fit items i in the data structure for Q on D it holds that:*

(a) the assignments yielded by $\textsc{Enum}(i, \{<^u\}_{u \in \mathsf{succ}(v^i)})$ *in Algorithm 3 are exactly the assignments in $\widetilde{\mathcal{E}}^i$.*

(b) The procedure $\textsc{Enum}(i, \{<^u\}_{u \in \mathsf{succ}(v^i)})$ *in Algorithm 3 takes time $O(|\,\mathsf{succ}(v^i)|)$*
- *until the first assignment will be yielded and*
- *between two assignments were yielded and*
- *the last assignment will be yielded and the procedure finished.*

(c) $\textsc{Enum}(i, \{<^u\}_{u \in \mathsf{succ}(v^i)})$ *in Algorithm 3 does not output duplicates.*

Proof. Let T_Q be the q-tree of Q that is used to construct the data structure for Q on D and let T_{free} be the subtree of T_Q induced on $\text{free}(Q)$.

Algorithm 3 Enumeration algorithm.

```
1: function ENUM(i, {<^v}_{v∈succ(v^i)})
2:     Input: present and fit item i in the data structure D and a set that contains
       for every w ∈ succ(v) in T_free an order over their children.
3:     if v^i is a leaf in T_free then
4:         yield α^i
5:     else
6:         Let u_1 <^{v^i} ... <^{v^i} u_s be the chilren of v^i that belongs to free(Q).
7:         for î_1 ∈ L^i_{u_1} do
8:             for α_1 ∈ ENUM(î_1, {<^v}_{v∈succ(u_1)}) do
9:                 ⋱
10:                for î_s ∈ L^i_{u_s} do
11:                    for α_s ∈ ENUM(î_s, {<^v}_{v∈succ(u_s)})) do
12:                        yield α^i ∪ α_1 ∪ ··· ∪ α_s
13:
14: if [∅] is fit then
15:     for α ∈ ENUM([∅], {<^v}_{v∈V'}) do
16:         print (α(x_1), ..., α(x_k), b_1, ..., b_s)
17: print EOE
```

We show Lemma 4.14 by induction of the height of an item in T_{free}.

For the induction base let us consider an item i of height 0, i.e., v^i is a leaf in T_{free}. Then, we simply output α^i. This is exactly the only assignment in $\widetilde{\mathcal{E}}^i$ and it takes $O(1)$ time until the assignment will be yield and until the procedure will finish. Clearly, we do not output duplicates.

For the inductive step let us consider an item of height h. Let $u_1, \cdots, u_s \subseteq \text{free}(Q)$ be the children of v^i in T_{free}. By construction of the algorithm it holds that for all $j \in [s]$ there is a $\widehat{\iota}_j \in \mathcal{L}^i_{u_j}$ such that $\alpha^i \cup \alpha_1 \cup \cdots \cup \alpha_s$ will be yield, where α_j will be yielded by $\text{ENUM}(\widehat{\iota}_j)$. Note that the items $\widehat{\iota}_j$ for all $j \in [k]$ are fit. Therefore, by induction hypothesis, these are exactly the assignments in $\widetilde{\mathcal{E}}^{\widehat{\iota}_j}$. Thus, with Lemma 4.12 it follows that the assignments, yielded by $\text{ENUM}(i, \{<^u\}_{u \in \mathsf{succ}(v^i)})$ are exactly the assignments in $\widetilde{\mathcal{E}}^i$.

Part (b) follows from the fact that for every iteration of the for-loop, where we iterate over assignments in $\text{ENUM}(\hat{\iota}_j, \{<^w\}_{w \in \mathsf{succ}(u_j)})$, it takes by induction hypothesis time $O(|\,\mathsf{succ}(u)|)$ to receive the first assignment we consider, and between two considered assignments, and between the last assignment and the end of the for-loop. Furthermore, we need time $O(1)$ to go to the first or the next element of $\mathcal{L}^i_{u_j}$. Therefore by induction hypothesis, we need $O(\sum_{j=1}^{s} |\,\mathsf{succ}(u_j)|) = O(\mathsf{succ}(v^i))$ until the first assignment will be yielded and between two assignments were yielded and between the last assignment will be yielded and the end of the procedure.

To prove part (c), let us assume for a contradiction, that an assignment α was yielded twice. We consider now the outermost loop, that continued with the iteration

between the two times α was yielded. If the loop is of the form $\widehat{\iota} \in \mathcal{L}^i_{u_j}$, item $\texttt{[}\alpha^i \frac{\alpha(u_j)}{u_j}\texttt{]}$ was considered twice. This is a contradiction to the construction of the data structure. If the loop is of the form $\alpha_j \in \mathsf{Enum}(\widehat{\iota}_j)$, then α_j was considered twice but this is a contradiction to the induction hypothesis. □

Algorithm 3 is correct, since

$$\begin{aligned} Q(D) &= \{(\alpha(x_1), \ldots, \alpha(x_k), b_1, \ldots, b_\ell) \ : \ \alpha \in \widetilde{\mathcal{E}}^{\texttt{[}\emptyset\texttt{]}}\} \\ &\overset{4.14(a)}{=} \{(\alpha(x_1), \ldots, \alpha(x_k), b_1, \ldots, b_\ell) : \alpha \textbf{ in } \textsc{Enum}(\texttt{[}\emptyset\texttt{]}, \{<^v\}_{v \in V'})\}. \end{aligned}$$

Since we can check in time $O(1)$ (using the Boolean variable `start-is-fit` in the data structure) if $\texttt{[}\emptyset\texttt{]}$ is fit and Lemma 4.14(b), we enumerate the tuples with delay $t_d = O(|\text{vars}(Q)|)$ without repetition (this is guaranteed by Lemma 4.14(c)).

This concludes the proof of Theorem 3.3(b).

Remark 4.15. *As a remark, we show now that if the data structure is lexicographically ordered. Then the enumeration algorithm* $\textsc{Enum}(\texttt{[}\emptyset\texttt{]}, \{<^v_s\}_{v \in V'})$ *where* $x_j <^v_s x_{j'}$ *if and only if* $j < j'$ *and* $x_j, x_{j'} \in child(v)$ *for all* $v \in V$*, enumerates the tuples in lexicographical order.*

Let $(\overline{a}_1, \overline{b}) \in Q(D)$ *and* $(\overline{a}_2, \overline{b}) \in Q(D)$ *where* $\overline{b} = (b_1, \ldots, b_\ell)$ *be two tuples such that* $\overline{a}_1$ *will be enumerated before* $\overline{a}_2$*. Let* $j \in [k]$ *be the smallest index with* $(\overline{a}_1)_j \neq (\overline{a}_2)_j$*. By definition of* $\{<^v_s\}_{v \in V'}$ *is the outermost loop, that continued with the iteration between* $\overline{a}_1$ *and* $\overline{a}_2$ *were enumerated in the for loop of the form* $\widehat{\iota} \in \mathcal{L}^i_{x_j}$*. Since these lists are lexicographically ordered, it follows that* $(\overline{a}_1)_j < (\overline{a}_2)_j$ *and thus* $\overline{a}_1 < \overline{a}_2$*.*

4.7. Enumerating the difference

The aim of this section is to show how to enumerate the tuples that are additionally in the result set and the tuples that do not belong to the result set any more after the database received some update steps, i.e., this chapter is devoted to prove Theorem 3.3 (c). For the remainder of this section we assume that $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is a q-hierarchical conjunctive query, $\text{vars}(Q) = \{x_1, \ldots, x_m\}$ with $0 \leqslant k \leqslant m$, and Q is of the form

$$Q \quad = \quad \left\{(x_1 \ldots x_k, b_1, \cdots, b_\ell) \ : \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right)\right\}, \tag{4.5}$$

where $b_1, \ldots, b_\ell \in \textbf{dom}$ and $\psi_1, \ldots, \psi_d$ are atomic queries of schema σ.

In the scenario here we need the system at one point to record the differences of the result set and after a couple of update steps we have the system start the enumeration of the difference. In this section we show how to enrich the data structure such that we can enumerate the difference with constant delay. Let D^- be the database at the start of the recording and let D^+ be the database at the time we want to enumerate the difference, i.e., D^+ is the database obtained from D^- after the updates were processed. The aim is to enumerate the set $Q(D^+) \setminus Q(D^-)$ (the "new" tuples in the result set)

and $Q(D^-) \setminus Q(D^+)$ (the tuples that are not in the result any more). We call this routine the **diff**-routine.

The task of enumerating the difference has been considered in various papers in the sense of delta queries [59, 66, 67, 81]. A delta query is defined as a query whose result contains tuples that are additionally in the result and do not belong in the result of Q on D any more after the update. In [59] an update step might involve insertions and deletions of multiple tuples to the database instead of single tuples as stated in this thesis. Such updates can be simulated in the following way. Before proceeding an update step, we need the system to record the difference, and then to insert or delete the tuples into/from the database one by one. Note that here we use set semantics instead of bag semantics as in [59]. As a warm up, let us consider the following example.

Example 4.16. *As an example, let us consider the query Q and the database D_0 in Example 4.4 as D^-. We need the system that to record the difference and receive the updates* delete $F(2,2,4)$, *insert* $E(3,4)$, *insert* $F(3,1,2)$, *insert* $G(3,1,2)$. *Let D^+ be the database we obtain after applying the updates and we then aim to start the enumeration of the difference. See Figure 4.6 for an illustration of D^+. The items that become fit after the update are marked yellow and the item that loses the fit status are marked red.*

Note that the reason why there are tuples in $Q(D^+) \setminus Q(D^-)$ is that new tuples were inserted to relations. In particular, we have $a_0 = 3$ and ($a_1 = 4$ or ($a_2 = 1$ and $a_3 = 2$)) for all $(a_0, a_1, a_2, a_3) \in Q(D^+) \setminus Q(D^-)$. This first idea is to enumerate the tuples $(a_0, a_1, a_2, a_3) \in Q(D^+)$ with $a_0 = 3$ and $a_1 = 4$ and afterwards the tuples with $a_0 = 3$ and $a_2 = 1$ and $a_3 = 2$. But then we would enumerate the tuple $(3, 4, 1, 2)$ twice. To get rid of this problem, we have to take into account that we enumerate in the second round only tuples with $a_1 \neq 4$.

To enumerate $Q(D^-) \setminus Q(D^+)$ we only have to deal with tuples we removed from the relation and enumerate the tuples $(a_0, a_1, a_2, a_3) \in Q(D^-)$ with $a_0 = a_2 = 2$ and $a_3 = 4$.

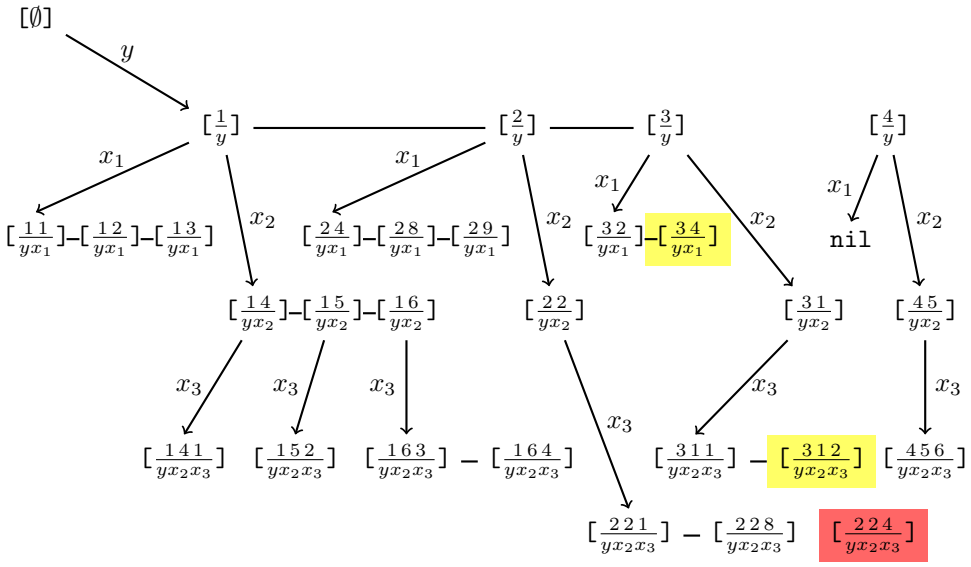
As we see in Example 4.16 here, there are some open questions for enumerating the difference.

1. How to find the insertion (deletion) items that force new tuples being in $Q(D^+) \setminus Q(D^-)$ or $Q(D^-) \setminus Q(D^+)$?
2. How to ensure that we do not enumerate a tuple twice?

To answer question 1, note that it is not sufficient to simply iterate over all insertion steps and then iterate over the new tuples forced by the insertion. For instance, if we insert in our example database tuples $(5,1), \ldots, (5,100)$ to E, the difference $Q(D^+) \setminus Q(D^-)$ is still empty but we can not enumerate this with constant delay. Thus, we use the fact, that whenever we have to take the insertion of a tuple to a relation into account for the difference, at least one item gets fit in the data structure.

To find these items efficiently we additionally store for every present item i in the data structure Boolean variables $\texttt{fit}^+[i]$ and $\texttt{fit}^-[i]$ and for every child u of v^i two

Figure 4.6.: Illustration from D^+.



lists in the data structure $(\mathcal{L}_u^i)^+$ and $(\mathcal{L}_u^i)^-$. By default, the Boolean variables will be initialised to false and the lists will be initialised as empty lists. The aim is that the following for the Boolean variables and the lists holds.

$$\texttt{fit}^\circ[i] \text{ is } \texttt{true} \iff i \text{ is fit in } D^\circ \text{ and not in } D^\otimes \tag{4.6}$$

and

$$(\mathcal{L}_u^i)^\circ := \left\{ \iota : \begin{array}{l} \iota \in \mathcal{L}_u^i \text{ in } D^\circ \text{ and there is a } \beta \supseteq \alpha^\iota \text{ such that } \texttt{fit}^\circ[\llbracket\beta\rrbracket] \text{ is } \texttt{true} \\ \text{and for all } w \in \text{vpath}[v^{\llbracket\beta\rrbracket}] \setminus \text{vpath}[v^\iota) \text{ is } \llbracket\beta|_{\text{vpath}[w]}\rrbracket \text{ fit in } D^\circ \end{array} \right\} \tag{4.7}$$

for $\circ \in \{+,-\}$ and $\otimes \in \{+,-\} \setminus \{\circ\}$. Intuitively, an item ι belongs to a list $(\mathcal{L}_u^i)^\circ$ if and only if there is a path from ι to an item that is fit in D° and not in $D^\otimes$. These lists will help us to find these items that are sufficient to enumerate the difference.

Example 4.17. *Consider the setting of Example 4.16. Then* $\texttt{fit}^+[\llbracket \frac{3\ 4}{yx_1} \rrbracket] = \texttt{fit}^+[\llbracket \frac{3\ 1\ 2}{yx_2x_3} \rrbracket] =$ `True` *and for the other items* i *we have* $\texttt{fit}^+[i] =$ `False`. $\texttt{fit}^-[\llbracket \frac{2\ 2\ 4}{yx_2x_3} \rrbracket] =$ `True` *and for the other items* i *is* $\texttt{fit}^-[i] =$ `False`. *Furthermore,* $(\mathcal{L}_y^{\llbracket\emptyset\rrbracket})^+ = \left\{ \llbracket \frac{3}{y} \rrbracket \right\}$, $(\mathcal{L}_{x_1}^{\llbracket \frac{3}{y} \rrbracket})^+ = \left\{ \llbracket \frac{3\ 4}{yx_1} \rrbracket \right\}$, $(\mathcal{L}_{x_2}^{\llbracket \frac{3}{y} \rrbracket})^+ = \left\{ \llbracket \frac{3\ 1}{yx_2} \rrbracket \right\}$ *and* $(\mathcal{L}_{x_3}^{\llbracket \frac{3\ 1}{yx_2} \rrbracket})^+ = \left\{ \llbracket \frac{3\ 1\ 2}{yx_2x_3} \rrbracket \right\}$. *For all the other items* ι *and for all* $u \in \text{child}(v^\iota)$ *it is* $(\mathcal{L}_u^\iota)^+ = \emptyset$. *Moreover,* $(\mathcal{L}_y^{\llbracket\emptyset\rrbracket})^- = \left\{ \llbracket \frac{2}{y} \rrbracket \right\}$, $(\mathcal{L}_{x_2}^{\llbracket \frac{2}{y} \rrbracket})^- = \left\{ \llbracket \frac{2\ 2}{yx_2} \rrbracket \right\}$ *and* $(\mathcal{L}_{x_3}^{\llbracket \frac{2\ 2}{yx_2} \rrbracket})^- = \left\{ \llbracket \frac{2\ 2\ 4}{yx_2x_3} \rrbracket \right\}$. *For all the other items* ι *and for all* $u \in \text{child}(v^\iota)$ *is* $(\mathcal{L}_u^\iota)^- = \emptyset$. *As we see here, when following*

the $(\mathcal{L}_u^i)^\circ$ *sets, we obtain a path from* $(\mathcal{L}_y^{[\emptyset]})^\circ$ *to the items* i *where* $\texttt{fit}^\circ[i]$ *is* true *for* $\circ \in \{+,-\}$.

The list $(\mathcal{L}_u^i)^-$ will be used to enumerate the tuples in $Q(D^-) \setminus Q(D^+)$ since the fact that an item loses the fit-status implies that new tuples are in the result set. The idea is the following. An algorithm enumerates for $\circ \in \{+,-\}$ the items i where $\texttt{fit}^\circ[i]$ is true, using the $(\mathcal{L}_u^i)^\circ$-lists. Then we take for each such item i the assignment α^i and enumerate the tuples $(\beta(x_1),\dots,\beta(x_k),b_1,\dots,b_\ell)$ in $Q(D^\circ)$ where $\alpha^i \subseteq \beta$, i.e., the output tuple coincides with the assignment of the item i. The next claim shows that these items help us to compute the tuples that belong to the difference set $Q(D^\circ) \setminus Q(D^\otimes)$ where $\otimes \in \{+,-\} \setminus \{\circ\}$.

Claim 4.18. *Let* Q *be a q-hierarchical query and let* D^- *be a database and* D^+ *be a database obtained from* D^- *after some updates on* D^-. *For every assignment* β *with* $(D^\circ,\beta) \models Q$ *the following holds.*

$$(\beta(x_1),\dots,\beta(x_k),b_1,\dots,b_\ell) \in Q(D^\circ) \setminus Q(D^\otimes) \iff$$

there is an item i *with* $\alpha^i \subseteq \beta$ *such that for all* $w \in \text{free}(Q)$, *is* $[\![\beta|_{\text{vpath}[w]}]\!]$ *fit in* D° *and* $\texttt{fit}^\circ[i]$ *is* true

Proof.

$$\begin{aligned}
&(\beta(x_1),\dots,\beta(x_k),b_1,\dots,b_\ell) \in Q(D^\circ) \setminus Q(D^\otimes)\\
\iff\ & (\beta(x_1),\dots,\beta(x_k),b_1,\dots,b_\ell) \in Q(D^\circ) \text{ and}\\
& (\beta(x_1),\dots,\beta(x_k),b_1,\dots,b_\ell) \notin Q(D^\otimes)\\
\iff\ & \text{for all } \psi \in \text{atoms}(Q) \text{ it holds } (D^\circ,\beta) \models \psi \text{ and}\\
& \text{there is a } \psi \in \text{atoms}(Q) \text{ such that } (D^\otimes,\beta) \not\models \psi\\
\iff\ & \text{for all } \psi \in \text{atoms}(Q) \text{ it holds } (D^\circ,\beta) \models \psi \text{ and}\\
& \text{there is a } \psi \in \text{atoms}(Q) \text{ such that } (D^\otimes,\beta|_{\text{vars}(\psi)}) \not\models \psi\\
\overset{\text{Def. fit}}{\iff}\ & \text{for all } w \in \text{free}(Q) \text{ is } [\![\beta|_{\text{vpath}[w]}]\!] \text{ fit in } D^\circ \text{ and}\\
& [\![\beta|_{\text{vars}(\psi)}]\!] \text{ is not fit in } D^\otimes\\
\iff\ & \text{for all } w \in \text{free}(Q) \text{ it holds } [\![\beta|_{\text{vpath}[w]}]\!] \text{ is fit in } D^\circ\\
& \text{and there is an item } i \text{ in } D^\circ \text{ with } \alpha^i = \beta|_{\text{vars}(\psi)} \subseteq \beta\\
& \text{and } \texttt{fit}^\circ[i] \text{ is true}
\end{aligned}$$

□

We will use the following claims to test efficiently if an item has to belong to a list $(\mathcal{L}_u^\iota)^\circ$.

Claim 4.19. *For all* $\circ \in \{+,-\}$, *for all present items* ι *and their parent item* $i = \alpha^\iota|_{\text{vpath}[v^\iota)}$ *the following holds:* $\iota \in (\mathcal{L}_u^i)^\circ$ *if and only if* $\iota \in \mathcal{L}_u^i$ *in* D° *and* $\texttt{fit}^\circ[\iota]$ *is* true *or there is a* $w \in \mathit{child}(v^\iota)$ *such that* $(\mathcal{L}_w^\iota)^\circ \neq \emptyset$.

Proof. • For the "if"-direction let $\iota \in (\mathcal{L}_u^i)^\circ$. Clearly, $\iota \in \mathcal{L}_u^i$ in D°. Furthermore, there is a $\beta \supseteq \alpha^\iota$ such that $\texttt{fit}^\circ[\![\beta]\!]$ is true and for all $w \in \text{vpath}[v^{[\![\beta]\!]}] \setminus \text{vpath}[v^\iota)$ is $[\![\beta|_{\text{vpath}[w]}]\!]$ fit in D°.

  - If $\beta = \alpha^\iota$, the claim holds since $\texttt{fit}^\circ[\iota]$ is true.
  - Otherwise, let $w \in \mathit{child}(u)$ be the child such that $w \in \mathrm{dom}(\beta)$ and let $\gamma := \beta|_{\mathrm{vpath}[w]}$. Since γ is fit in D°, it holds that $\texttt{[}\gamma\texttt{]} \in \mathcal{L}^\iota_w$ in D° and there is a $\beta \supseteq \gamma$ such that $\texttt{fit}^\circ[\texttt{[}\beta\texttt{]}]$ is true and for all $w \in \mathrm{vpath}[v^{\texttt{[}\beta\texttt{]}}] \setminus \mathrm{vpath}[\gamma)$ is $\texttt{[}\beta|_{\mathrm{vpath}[w]}\texttt{]}$ fit in D°. Therefore, $\texttt{[}\gamma\texttt{]} \in (\mathcal{L}^\iota_w)^\circ$ and, in particular, $(\mathcal{L}^\iota_w)^\circ \neq \emptyset$.

- For the "only if"-direction let us consider that $\iota \in \mathcal{L}^i_u$ in D°, and $\texttt{fit}^\circ[\iota]$ is true or there is a $w \in \mathit{child}(v^\iota)$ such that $(\mathcal{L}^\iota_w)^\circ \neq \emptyset$.
  - If $\texttt{fit}^\circ[\iota]$ is true, the claim follows by the definition of $(\mathcal{L}^i_u)^\circ$.
  - Otherwise there is a $w \in \mathit{child}(v^\iota)$ such that $(\mathcal{L}^\iota_w)^\circ \neq \emptyset$. Let $\widehat{\iota} \in (\mathcal{L}^\iota_w)^\circ$ arbitrary. Then, there is a $\beta \supseteq \alpha^{\widehat{\iota}}$ such that $\texttt{fit}^\circ[\texttt{[}\beta\texttt{]}]$ is true for all $w \in \mathrm{vpath}[v^{\texttt{[}\beta\texttt{]}}] \setminus \mathrm{vpath}[v^{\widehat{\iota}})$ we have that $\texttt{[}\beta|_{\mathrm{vpath}[w]}\texttt{]}$ is fit in D°. Since $\iota \in \mathcal{L}^i_u$ in D°, ι is fit in D° and we obtain $\beta \supseteq \alpha^\iota$ and for all $w \in \mathrm{vpath}[v^{\texttt{[}\beta\texttt{]}}] \setminus \mathrm{vpath}[v^\iota)$ we have that $\texttt{[}\beta|_{\mathrm{vpath}[w]}\texttt{]}$ is fit in D°. Thus, $\iota \in (\mathcal{L}^i_u)^\circ$.

□

Claim 4.20. *For all $\otimes \in \{+,-\}$, for all present items ι and their parent item $i = \alpha^\iota|_{\mathrm{vpath}[v^\iota)}$ the following holds: It takes time $O(|\mathit{child}(v^\iota)|)$ to test whether $\iota \in (\mathcal{L}^i_u)^\otimes$.*

Proof. To test if $\iota \in (\mathcal{L}^i_u)^\otimes$ is correct, it is sufficient by Claim 4.20, to test if ι is fit in D° and $\texttt{fit}^\circ[\iota]$ is true or if there is a $w \in \mathit{child}(v^\iota)$ such that $(\mathcal{L}^\iota_w)^\circ$ is empty.

To test if ι is fit in D^+, we simply test if it is fit in the current data structure. To test if ι is fit in D^-, we test if $\texttt{fit}^-[i]$ is true or ι is fit in the current data structure and $\texttt{fit}^+[i]$ is false. By definition of $\texttt{fit}^\circ[i]$ we obtain the information if ι is fit in D^-. Thus, the test can be done in $O(|\mathrm{vars}(Q)|)$. □

In the following Remark 4.21 it is described how to efficiently maintain the lists and to efficiently clean up the data structure, i.e., to obtain the data structure that we would have if we did not start the difference procedure.

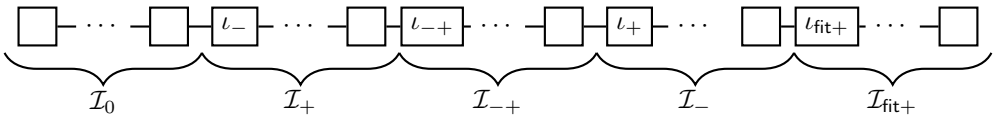
Remark 4.21. *Here we describe how to maintain the lists such that we can efficiently enumerate the items of the lists and how to clean up the data structure after the difference routine.*

To avoid notational clutter, we define the following sets.

$$
\begin{aligned}
\mathcal{I}_0 &:= \mathcal{L}^i_u \setminus ((\mathcal{L}^i_u)^+ \cup (\mathcal{L}^i_u)^-) \\
\mathcal{I}_- &:= (\mathcal{L}^i_u)^- \setminus (\{\iota \in (\mathcal{L}^i_u)^- \ : \ \texttt{fit}^-[i] \textit{ is } \mathsf{true}\} \cup (\mathcal{L}^i_u)^+) \\
\mathcal{I}_{-+} &:= (\mathcal{L}^i_u)^- \cap (\mathcal{L}^i_u)^+ \\
\mathcal{I}_+ &:= (\mathcal{L}^i_u)^+ \setminus (\{\iota \in (\mathcal{L}^i_u)^+ \ : \ \texttt{fit}^+[i] \textit{ is } \mathsf{true}\} \cup (\mathcal{L}^i_u)^-) \\
\mathcal{I}_{\mathsf{fit}+} &:= \{\iota \in (\mathcal{L}^i_u)^+ \ : \ \texttt{fit}^+[i] \textit{ is } \mathsf{true}\} \\
\mathcal{I}_{\mathsf{fit}-} &:= \{\iota \in (\mathcal{L}^i_u)^- \ : \ \texttt{fit}^-[i] \textit{ is } \mathsf{true}\}
\end{aligned}
$$

It is straightforward to verify that the sets above are pairwise disjoint. Note that every item in $\iota \in \mathcal{I}_{-+}$ is fit in D^+ and D^- (by Definition of $(\mathcal{L}^i_u)^-$ and $(\mathcal{L}^i_u)^+$)and thus by definition of $\texttt{fit}^\circ[\iota]$ for $\circ \in \{+,-\}$, it follows that $\texttt{fit}^+[\iota] = \texttt{fit}^-[\iota] = \mathsf{false}$.

Figure 4.7.: Illustration of $\mathcal{L}_u^i$.



Furthermore, it holds $(\mathcal{L}_u^i)^+ \subseteq \mathcal{L}_u^i$ *since every item that is fit in* D^+ *is fit in the current database. Moreover,* $(\mathcal{L}_u^i)^- \setminus \mathcal{I}_{\mathsf{fit}-} \subseteq \mathcal{L}_u^i$ *since the items in* $(\mathcal{L}_u^i)^- \setminus \mathcal{I}_{\mathsf{fit}-}$ *are fit in* D^+ *and in* D^-*. Using these facts it is straightforward to verify that*

$$\mathcal{L}_u^i \;=\; \mathcal{I}_0 \,\cup\, \mathcal{I}_- \,\cup\, \mathcal{I}_{-+} \,\cup\, \mathcal{I}_+ \,\cup\, \mathcal{I}_{\mathsf{fit}+}$$

We organize the list $\mathcal{L}_u^i$ *by the following way.*

Let $<$ *be the linear order of the item in the list* $\mathcal{L}_u^i$*. We store the list* $\mathcal{L}_u^i$ *as a concatenation of* $\mathcal{I}_0, \mathcal{I}_-, \mathcal{I}_{-+}, \mathcal{I}_+$ *and* $\mathcal{I}_{\mathsf{fit}+}$ *(in this order). For all* $\circ \in \{+, -+, -, \mathsf{fit}+\}$ *let* $\iota_\circ$ *be the first element of* $\mathcal{I}_\circ$ *if it is not empty, i.e.,*

$$\iota_\circ \;=\; \begin{cases} \textit{the first item in } \mathcal{I}_\circ & \textit{if } \mathcal{I}_\circ \neq \emptyset \\ \texttt{nil} & \textit{if } \mathcal{I}_\circ = \emptyset \end{cases} \tag{4.8}$$

We call these items separator items.

In other words, if $\iota_\circ \neq \texttt{nil}$ *for all* $\circ \in \{+, -+, -, \mathsf{fit}+\}$*, the items in* $\mathcal{L}_u^i$ *are organized by the following way.*

- *The items in* $\iota \in \mathcal{L}_u^i$ *with* $\iota < \iota_-$ *belong to* $\mathcal{I}_0$*.*
- *The items in* $\iota \in \mathcal{L}_u^i$ *with* $\iota_- \leqslant \iota < \iota_{-+}$ *belong to* $\mathcal{I}_-$*.*
- *The items in* $\iota \in \mathcal{L}_u^i$ *with* $\iota_{-+} \leqslant \iota < \iota_+$ *belong to* $\mathcal{I}_{-+}$*.*
- *The items in* $\iota \in \mathcal{L}_u^i$ *with* $\iota_+ \leqslant \iota < \iota_{\mathsf{fit}+}$ *belong to* $\mathcal{I}_+$*.*
- *The items in* $\iota \in \mathcal{L}_u^i$ *with* $\iota_{\mathsf{fit}+} \leqslant \iota$ *belong to* $\mathcal{I}_{\mathsf{fit}+}$*.*

See Figure 4.7 for an illustration.

To maintain the order above, we use Algorithm 4 whenever we insert an item ι *to* $\mathcal{L}_u^i$ *and Algorithm 5 whenever we delete an item* ι *from* $\mathcal{L}_u^i$*. If an item modifies the membership to one of the sets* $\mathcal{I}_\circ$*, we use Algorithm 5 to delete* ι *from* $\mathcal{L}_u^i$ *and afterwards Algorithm 4 to insert the item* ι *to* $\mathcal{L}_u^i$*. In these algorithms we let* $- < -+ < + < \mathsf{fit}+$*.*

It is straightforward to verify that we have the order described above if an item is inserted to the list $\mathcal{L}_u^i$*, deleted from the list or the membership from one of the* $\mathcal{I}_\circ$ *is modified.*

To test whether an item ι *belongs to a list* $(\mathcal{L}_u^i)^\circ$*, we use Claim 4.20. This can be used to get the element* $\circ \in \{0, -, -+, +, \mathsf{fit}+, \mathsf{fit}-\}$ *for which* $\iota \in \mathcal{I}_\circ$*. Thus, line 1*

Algorithm 4 An item ι will be inserted to $\mathcal{L}_u^i$.

1: Test for which $\circ \in \{0, -, -+, +, \mathsf{fit}+, \mathsf{fit}-\}$ it holds $\iota \in \mathcal{I}_\circ$. Let $\circ$ be the corresponding element.
2: **if** $\iota \in \mathcal{I}_{\mathsf{fit}-}$ **then**
3: Insert ι to the list $\mathcal{I}_{\mathsf{fit}-}$.
4: **if** $\iota_\circ \neq \texttt{nil}$ **then**
5: Insert ι as the successor of $\iota_\circ$.
6: **else**
7: $\iota_\circ := \iota$
8: **if** there is a $\oplus \in \{-, -+, +, \mathsf{fit}+\}$ with $\circ < \oplus$ and $\iota_\oplus \neq \texttt{nil}$ **then**
9: Let $\otimes = \min\{\oplus \in \{-, -+, +, \mathsf{fit}+\} \,:\, \circ < \oplus,\ \iota_\oplus \neq \texttt{nil}\}$
10: Insert ι as the previous element of $\iota_\otimes$, i.e., as the successor element of the previous of $\iota_\otimes$ if it exists or as the first element in the list otherwise.
11: **else**
12: Insert ι as the last element of $\mathcal{L}_u^i$.

Algorithm 5 An item ι will be deleted from $\mathcal{L}_u^i$.

1: **if** there is a $\circ \in \{-, -+, +, \mathsf{fit}+\}$ such that $\iota_\circ = \iota$ **then**
2: **if** the successor element is an separator item, i.e., an item $\iota_\oplus$ with $\oplus \in \{-+, +, \mathsf{fit}+\}$ **then**
3: $\iota_\circ := \texttt{nil}$.
4: **else**
5: Define $\iota_\circ$ as the successor item of ι.
6: Delete ι from $\mathcal{L}_u^i$.

can be proceed in time $O(poly(Q))$. The other lines in Algorithm 4 and all lines in Algorithm 5 take time $O(1)$. Thus, it takes time $O(poly(Q))$ to insert or delete an element in the list.

In the remainder of this section it is necessary to efficiently enumerate the elements in $(\mathcal{L}_u^i)^-$, $(\mathcal{L}_u^i)^+$ and $\mathcal{L}_u^i \setminus \mathcal{I}_{\mathsf{fit}+}$. This can be done with the separator items:

- *To enumerate $(\mathcal{L}_u^i)^-$, start with ι_- (or ι_{-+} if ι_- is $\mathtt{nil}$) and iterate through the elements until an item $\iota_\circ$ with $\circ \in \{+, \mathsf{fit}+\}$ is reached. If $\iota_- = \iota_{-+} = \mathtt{nil}$, then $(\mathcal{L}_u^i)^- \setminus \mathcal{I}_{\mathsf{fit}-} = \emptyset$. Afterwards, we enumerate the set $\mathcal{I}_{\mathsf{fit}-}$.*
- *To enumerate $(\mathcal{L}_u^i)^+$, start with ι_{-+} (or ι_+ if ι_{-+} is $\mathtt{nil}$) and iterate through the elements until the end.*
- *To enumerate $\mathcal{L}_u^i \setminus \mathcal{I}_{\mathsf{fit}+}$, start with the first item of $\mathcal{L}_u^i$ and iterate until we reach $\iota_{\mathsf{fit}+}$ or, if $\iota_{\mathsf{fit}+}$ is $\mathtt{nil}$, until the end of the list.*

For every separator item ι, we store a pointer to ι together with a timestamp. The timestamp is equal to the number of times we request the enumeration of the difference.

The timestamp reveals if ι is the current separator item. A separator item is not $\mathtt{nil}$ if and only if the timestamp equals the current timestamp. If we want to set the new separator item, we overwrite the timestamp. We use the same trick for the timestamps of $\mathtt{fit}^\circ[i]$. Instead of $\mathtt{fit}^\circ[i] = \mathsf{true}$ we store the last timestamp where we have set this value to true. Then, we know that $\mathtt{fit}^\circ[i]$ is set to true if and only if $\mathtt{fit}^\circ[i]$ has the current timestamp. This is convenient to clean up the data structure after enumerating the difference, i.e., to recover the data structure that we would obtain if we did not use the difference method. When resetting the difference procedure, we obtain that the timestamps get old and $\mathtt{fit}^\circ[i] = \mathsf{false}$ for all items i. Furthermore, we have that the separator items are not the current separator items when the timestamps are getting old and the list $\mathcal{I}_\circ$ is automatically empty for all $\circ \in \{-, -+, +, \mathsf{fit}+\}$. For the set $\mathcal{I}_{\mathsf{fit}-}$ we also store a timestamp. The list is only a valid list if the stored timestamp is equal to the current timestamp. If we start a new list, we update the timestamp with a new list. If the timestamp is old, the list can be identified as empty. To clean up the data structure, we us the following idea. We store a list for every item ι for that $\mathtt{fit}^-[\iota]$ is true. Whenever an item gets this status, we store the item at the end of the list and we remove the fist item if the timestamp is old. Note that since we insert the items at the end of the list, it is sorted with respect to the timestamps.

We say that the data structure is correct if the conditions for the Boolean variables in Equation 4.6 and the lists in Equation 4.7 holds.

To ensure that the data structure is correct we execute, whenever an item gets fit during an update step, Algorithm 6 with $\circ := +$ and, whenever an item loses the fit status during an update step, Algorithm 6 with $\circ := -$.

To show that Algorithm 6 is correct, we have to show that the algorithm ensures the correctness of the data structure.

Claim 4.22. *1. The data structure is correct if $D^+ = D^-$.*

Algorithm 6 The fit status of $\texttt{[}\frac{a_1\cdots a_m}{v_1\cdots v_m}\texttt{]}$ changed.

Input: $\circ \in \{+,-\}$, item $\texttt{[}\frac{a_1\cdots a_m}{v_1\cdots v_m}\texttt{]}$ whose fit status was changed.
Let $\otimes \in \{+,-\} \setminus \{\circ\}$
Let $\iota_j := \texttt{[}\frac{a_1\cdots a_j}{v_1\cdots v_j}\texttt{]}$ for all $j \in \{1,\ldots,m\}$.
if $\texttt{fit}^{\otimes}[i_m]$ is true **then**
    $\texttt{fit}^{\otimes}[i_m] \leftarrow$false
    for $j = m$ **to** 1 **do**
        if $i_j \in (\mathcal{L}_u^{i_{j-1}})^{\otimes}$ but it is not correct **then**
            Remove i_j from $(\mathcal{L}_u^{i_{j-1}})^{\otimes}$
else if $\texttt{fit}^{\circ}[i_m]$ is false **then**
    $\texttt{fit}^{\circ}[i_m] \leftarrow$true
    for $j = m$ **to** 1 **do**
        if i_j is fit in D° **then** insert i_j to $(\mathcal{L}_{v_{j-1}}^{i_{j-1}})^{\circ}$
        else break ▷ Stop the for-loop

2. *If the data structure is correct and an item gets fit (loses the fit status), Algorithm 6 with $\circ := +$ (with $\circ := -$, resp.) ensures that the data structure is still correct.*

Proof. Clearly, the claim holds for $D^+ = D^-$ since we initialise $(\mathcal{L}_u^i)^+$ and $(\mathcal{L}_u^i)^-$ as empty lists and $\texttt{fit}^+[i]$ and $\texttt{fit}^-[i]$ as false.

For the second part we consider that the data structure is correct and ...

1. ... an item $i_m = \texttt{[}\frac{a_1\cdots a_m}{v_1\cdots v_m}\texttt{]}$ gets fit. For all $j \in [m]$ let i_j be the item $\texttt{[}\frac{a_1\cdots a_j}{v_1\cdots v_j}\texttt{]}$.
    - *Case 1a):* The item i_m was fit in D^-. Before i_m gets fit, it was fit in D^- and not in D^+ and, in particular, $\texttt{fit}^-[i_m]$ is true and has to be changed to false. Because of this fact, we have to check for all $j \in [m]$ if i_j is still allowed to be in $(\mathcal{L}_{v_j}^{i_{j-1}})^-$. If not, we remove the item from the list. This is done by Algorithm 6 using $\circ := +$.
    - *Case 1b):* The item i_m was not fit in D^-. Before i_m gets fit, it was not fit in D^- and not in D^+ and, in particular, $\texttt{fit}^+[i_m]$ is false and has to be changed to true. Because of this fact, we have to add fit items i_j to $(\mathcal{L}_{v_j}^{i_{j-1}})^+$ if there is a path over the $+$-lists to i_m. This is done by Algorithm 6 using $\circ = +$.
2. ... an item $i_m = \texttt{[}\frac{a_1\cdots a_m}{v_1\cdots v_m}\texttt{]}$ loses the fit status. For all $j \in [m]$ let i_j be the item $\texttt{[}\frac{a_1\cdots a_j}{v_1\cdots v_j}\texttt{]}$.
    - *Case 2a):* The item i_m was fit in D^-. Before i_m loses the fit status, it was fit in D^- and in D^+ and, in particular, $\texttt{fit}^-[i_m]$ is false and has to be changed to true. Because of this fact, we have to add fit items i_j in D^- to $(\mathcal{L}_{v_j}^{i_{j-1}})^-$ if there is a path over the $-$-lists to i_m. This is done by Algorithm 6 using $\circ = -$.

- *Case 2b):* The item i_m was not fit in D^-. Before i_m loses the fit status, it was not fit in D^- but fit in D^+ and, in particular, $\texttt{fit}^+[i_m]$ is true and has to be changed to false. Because of this fact, we have to check for all $j \in [m]$ whether i_j is still allowed to be in $(\mathcal{L}_{v_j}^{i_{j-1}})^+$. If not, we remove the item from the list. This is done by Algorithm 6 using $\circ = -$.

Note that by the definition of the $\mathcal{L}^\circ$ list it follows that if the fit status of an item i changes, it can only change the membership of an item ι to the $(\mathcal{L}_{v^\iota}^{\hat{\iota}})^\circ$ list for $\circ \in \{+,-\}$ if $\alpha^\iota \subseteq \alpha^i$. Therefore, it is sufficient to consider only these items in the algorithm. □

Now, we show that updates can be efficiently done in constant time, i.e., Algorithm 6 takes constant time.

Claim 4.23. *Algorithm 6 takes time* $O(\text{vars}(Q)^2)$

Proof. Since $(\mathcal{L}_u^i)^-$ and $(\mathcal{L}_u^i)^+$ are lists, lookup, insert and remove from the list can be done in $O(|\text{vars}(Q)|)$ and changing the Boolean variables $\texttt{fit}^+[i]$ and $\texttt{fit}^-[i]$ can also be done in $O(1)$. When applying Claim 4.20, it takes time $O(|\text{vars}(Q)|)$. Note that the information whether there is a $w \in \mathit{child}(v^\iota)$ such that $(\mathcal{L}_w^\iota)^\circ$ is empty is correct since the for-loop goes from d to 1. Thus, the test can be done in $O(|\text{vars}(Q)|)$. All in all, Algorithm 6 takes time $O(|\text{vars}(Q)|^2)$. □

We now show how to use them to enumerate the difference.

We say that an item i is $\circ$-redundant if and only if there is a $\beta \subset \alpha^i$ such that $\texttt{fit}^\circ[[\beta]]$ is true for all $\circ \in \{+,-\}$. The main idea is to enumerate all non-$\circ$-redundant items i and every extension of α^i. A $\circ$-redundant item is not relevant, since for example if we have an item i and an item ι with $\alpha^i \subset \alpha^\iota$, then we enumerate every extension of α^ι when we enumerate every extension of α^i. Algorithm 7 shows how to enumerate with constant delay relevant non-$\circ$-redundant items.

Algorithm 7 Enumerate non-$\circ$-redundant items i with $\texttt{fit}^\circ[i]$.

```
1: procedure EnumFit(∘, i)
2:     Input: Present item i and ∘ ∈ {+, −}.
3:     if fit^∘[i] is true then yield i
4:     else
5:         for u ∈ child(v^i) do
6:             for ι ∈ (L_u^i)^∘ do
7:                 yield EnumFit(∘, ι)
8:     output EOE
```

The following claim states that the algorithm is correct.

Claim 4.24. *For all present items i in the data structure it holds: $[\beta]$ is yielded by* EnumFit$(\circ, [\emptyset])$ *in Algorithm 7 if and only if*

- $\texttt{fit}^\circ[[\beta]]$ *is* true,

- *$\llbracket\beta\rrbracket$ is not $\circ$-redundant and*
- *for all $w \in \mathrm{vpath}[v^{\llbracket\beta\rrbracket}]$ we have $\llbracket\beta|_{\mathrm{vpath}[w]}\rrbracket$ fit in D°.*

Proof.

$$\begin{aligned}
&\llbracket\beta\rrbracket \text{ is yielded by } \mathsf{EnumFit}(\circ, \llbracket\emptyset\rrbracket) \\
\iff\ & \text{there are items } i_0,\dots,i_m \text{ and } v_0,\dots,v_m \in \mathrm{vars}(Q) \text{ with } v_j = v^{i_j} \\
& \text{and } i_j \in (\mathcal{L}_{v_{j-1}}^{i_{j-1}})^\circ \text{ for all } j \in [m] \text{ such that } \texttt{fit}^\circ[\llbracket\beta\rrbracket] \text{ is } \mathsf{true} \\
& \text{and } \texttt{fit}^\circ[i_j] \text{ is } \mathsf{false} \text{ for all } j \in [d-1] \\
\iff\ & \text{for all } w \in \mathrm{vpath}[v^{\llbracket\beta\rrbracket}] \text{ is } \llbracket\beta|_{\mathrm{vpath}[w]}\rrbracket \text{ fit in } D^\circ \text{ and} \\
& \texttt{fit}^\circ[\llbracket\beta\rrbracket] \text{ is } \mathsf{true} \text{ and } \llbracket\beta\rrbracket \text{ is not } \circ\text{-redundant}
\end{aligned}$$

The first equation follows by the construction of the algorithm and the second equivalence from the definition of $(\mathcal{L}_u^i)^\circ$ and $\circ$-redundant. □

The following claim states that the items will be enumerated with constant delay.

Claim 4.25. *EnumFit$(\circ, \llbracket\emptyset\rrbracket)$ enumerates the items with delay $O(|\mathrm{vars}(Q)|^2)$.*

Proof. By Claim 4.19 we obtain that on input of an item ι that belongs to a set $(\mathcal{L}_u^i)^\circ$ we have that $\texttt{fit}^\circ[i]$ is true or there is a $w \in \mathit{child}(v^i)$ such that $(\mathcal{L}_w^i)^\circ \neq \emptyset$. In particular, the algorithm will, upon input of such an item, yield at least one item. Since the recursion depth is bounded by $|\mathrm{vars}(Q)|$ the delay is $O(|\mathrm{vars}(Q)|^2)$. If there is for the start-item $\llbracket\emptyset\rrbracket$ no $w \in \mathit{child}(v_{root})$ with $(\mathcal{L}_w^{\llbracket\emptyset\rrbracket})^\circ \neq \emptyset$, the algorithm takes time $|\mathit{child}(v_{root})|$. All the other items that $\mathsf{EnumFit}$ receives as input, belong to a list $(\mathcal{L}_u^i)^\circ$. □

To get rid of the problem that we might enumerate a tuple twice, the following definition will be convenient:

For all assignments α and all $\circ \in \{+,-\}$ let

$$D^\circ_{\mathrm{VAR}}(\alpha) \ := \ \min\left\{v^i \ : \ \texttt{fit}^\circ[i] \text{ is } \mathsf{true} \text{ and } i \text{ is not redundant and } \alpha^i \subseteq \alpha\right\}$$

if it exists and $D^\circ_{\mathrm{VAR}}(\alpha) \ := \ \max \mathrm{vars}(Q)$ otherwise.

Example 4.26. *Recall Example 4.16. Then $D^+_{VAR}(\llbracket\frac{3\ 4}{yx_1}\rrbracket) = x_1$, $D^+_{VAR}(\llbracket\frac{3\ 1}{yx_2}\rrbracket) = x_3$, $D^+_{VAR}(\llbracket\frac{3\ 1\ 2}{yx_2x_3}\rrbracket) = x_3$ and $D^+_{VAR}(\llbracket\frac{3\ 4\ 1\ 2}{yx_1x_2x_3}\rrbracket) = x_1$.*

The algorithm for enumerating the difference is the following:

Algorithm 8 Enumerate the difference.

```
1: function DiffEnum(i, ι̂, ∘)
2:     Let u_1 < ··· < u_ℓ be the children of v^i that belong to free(Q).
3:     if v^i is a leaf then
4:         yield α^i
5:     else
6:         for ι_1 ∈ Es°_{u_1}(i, ι̂) do
7:             for α_1 ∈ DiffEnum(ι_1, ι̂, ∘) do
8:                 ⋱
9:                 for ι_ℓ ∈ Es°_{u_ℓ}(i, ι̂) do
10:                    for α_ℓ ∈ DiffEnum(ι_ℓ, ι̂, ∘) do
11:                        yield α^i ∪ α_1 ∪ ··· ∪ α_ℓ
12:
13: for ι̂ ∈ EnumFit(∘, ⟦∅⟧) do
14:     for α ∈ DiffEnum(⟦∅⟧, ι̂, ∘) do
15:         print (α(x_1), ..., α(x_ℓ), b_1, ..., b_ℓ)
```

where

$$\mathrm{Es}^{\circ}_{v}(i,\widehat{\iota}) := \begin{cases} \{\, \llbracket \alpha|_{\mathrm{vpath}[v]} \rrbracket \,\} & \text{if } v \in \mathrm{path}[v^{\widehat{\iota}}], \\ \mathcal{L}^{i}_{v} \cup \{\iota \in (\mathcal{L}^{i}_{v})^{\circ} : \mathtt{fit}^{\circ}[\iota] = \mathsf{true}\} & \text{if } v \notin \mathrm{path}[v^{\widehat{\iota}}] \text{ and } v > D^{\circ}_{\mathrm{VAR}}(\alpha^{\iota}), \\ \mathcal{L}^{i}_{v} \setminus \{\iota \in (\mathcal{L}^{i}_{v})^{\circ} : \mathtt{fit}^{\circ}[\iota] = \mathsf{true}\} & \text{otherwise .} \end{cases}$$

The set $\mathrm{Es}^{\circ}_{v}(i,\widehat{\iota})$ will help us to avoid that a tuple will be enumerated twice.

Example 4.27. *Recall Example 4.16 and let* $\iota_1 := \llbracket \frac{3\ 4}{yx_1} \rrbracket$ *and* $\iota_2 := \llbracket \frac{3\ 1\ 2}{yx_2x_3} \rrbracket$*. The sets* $\mathrm{Es}^{+}_{v}(i,\iota_j)$ *enumerate tuples which coincide only with the assignments* α^{ι_j} *for* $j \in \{1,2\}$*. Since* $\mathrm{Es}^{+}_{y}(\llbracket \emptyset \rrbracket, \iota_1) = \left\{ \llbracket \frac{3}{y} \rrbracket \right\}$*, the enumeration will be forced to use the assignment* $\llbracket \frac{3}{y} \rrbracket$ *for the enumeration. Then,* $\mathrm{Es}^{+}_{x_1}(\llbracket \frac{3}{y} \rrbracket, \iota_1) = \left\{ \llbracket \frac{3\ 4}{yx_1} \rrbracket \right\}$*, and* $\mathrm{Es}^{+}_{x_2}(\llbracket \frac{3}{y} \rrbracket, \iota_1) = \mathcal{L}^{\llbracket \frac{3}{y} \rrbracket}_{x_2}$ *since* $x_2 > x_1 = D^{+}_{VAR}(\frac{3\ 4}{yx_1})$*. This ensures that all tuples were enumerated that coincide with* α^{ι_1}*. Let us now consider* ι_2*. Then,* $\mathrm{Es}^{+}_{y}(\llbracket \emptyset \rrbracket, \iota_2) = \left\{ \llbracket \frac{3}{y} \rrbracket \right\}$*,* $\mathrm{Es}^{+}_{x_1}(\llbracket \frac{3}{y} \rrbracket, \iota_2) = \mathcal{L}^{\llbracket \frac{3}{y} \rrbracket}_{x_1} \setminus \left\{ \iota \in (\mathcal{L}^{\llbracket \frac{3}{y} \rrbracket}_{x_2})^{\circ} : \mathtt{fit}^{\circ}[\iota] = \mathit{true} \right\} = \left\{ \llbracket \frac{3\ 2}{yx_1} \rrbracket \right\}$*, since* $\mathtt{fit}^{+}[\frac{3\ 4}{yx_1}]$ *is* true*. This ensures that the tuple* $(3,4,1,2)$ *will not be enumerated twice.*

The next set will help us to describe the set of assignments that will be yielded by the DiffEnum procedure. For all present items i and all assignments γ and all $\circ \in \{+,-\}$ let:

$$\left(\Delta^{i}_{\gamma}\right)^{\circ} := \left\{ \beta \supseteq \alpha^{i} \ : \begin{array}{c} \mathrm{dom}(\beta) = \bigcup_{\psi \in \mathrm{atoms}(v^i)} \mathrm{vars}(\psi), \\ (D^{\circ}, \beta) \models \psi \text{ for all } \psi \in \mathrm{atoms}(v^i), \\ \text{for all } v \in \mathrm{dom}(\beta) \cap \mathrm{dom}(\gamma) \text{ is } \beta(v) = \gamma(v), \\ D^{\circ}_{\mathrm{VAR}}(\beta) \geqslant v^{\llbracket \gamma \rrbracket} \end{array} \right\}$$

Here, γ is the assignment we take into account for the enumeration, i.e., the aim is to enumerate all assignments β with $\gamma \subseteq \beta$. The set $\left(\Delta^i_\gamma\right)^\circ$ is the set of assignments that are extensions of α^i that we consider, when enumerating with γ.

Example 4.28. *Let us consider the assignments of the two items that gets fits after the updates in Example 4.16, i.e.,* $\gamma_1 := \frac{3\ 4}{yx_1}$ *and* $\gamma_2 := \frac{3\ 1\ 2}{yx_2x_3}$ *and let* $i = \llbracket\frac{3}{y}\rrbracket$. *Then,* $\frac{3\ 4\ 1\ 2}{yx_1x_2x_3} \in \left(\Delta^i_{\gamma_1}\right)^\circ$ *and* $\frac{3\ 4\ 1\ 2}{yx_1x_2x_3} \notin \left(\Delta^i_{\gamma_2}\right)^\circ$ *since* $D^+_{VAR}(\llbracket\frac{3\ 4\ 1\ 2}{yx_1x_2x_3}\rrbracket) = x_1 < x_3 = D^+_{VAR}(\gamma_2)$. *Thus, when we first enumerate the tuples that coincide with* γ_1, *the tuple* $(3, 4, 1, 2)$ *appears and when we enumerate the tuples that coincides with* γ_2, *we do not enumerate* $(3, 4, 1, 2)$ *for a second time.*

We now show that this is the set of assignments yielded by the DiffEnum procedure:

Claim 4.29. *For all assignments* γ *with* $\mathrm{dom}(\gamma) = \mathrm{vpath}[v]$ *for a* $v \in \mathrm{vars}(Q)$ *and for all items* i, *where*

- *for all* $v \in \mathrm{dom}(\gamma) \cap \mathrm{dom}(\alpha^i)$ *we have* $\gamma(v) = \alpha^i(v)$,
- $D^\circ_{VAR}(\alpha^i) > v^{\llbracket\gamma\rrbracket}$ *and*
- i *is fit in* D°,

the following holds for all $\circ \in \{+, -\}$. DiffEnum$(i, \llbracket\gamma\rrbracket, \circ)$ *enumerates the set* $\left(\Delta^i_\gamma\right)^\circ$.

Proof. Let T_Q be the q-tree of Q from which the data structure was constructed from and let T_{free} be the induced subgraph of T_Q induced on free(Q). We show this claim by induction over the height of i in T_{free}. For the induction base, let us consider an item i where v^i is a leaf in the q-tree of Q. By assumption and since i is fit in D° it holds $\left(\Delta^i_\gamma\right)^\circ = \{\alpha^i\}$.

For the inductive step, let us consider an item of height > 0. Let $u_1, \ldots, u_s$ denote

the children of v^i in T_{free}. Then, for all assignments β the following holds.

$$\beta \in \left(\Delta^i_\gamma\right)^\circ \overset{\text{Def. } \left(\Delta^i_\gamma\right)^\circ}{\iff} \begin{array}{c} \beta \supseteq \alpha^i,\ \mathrm{dom}(\beta) = \bigcup_{\psi \in \mathrm{atoms}(v^i)} \mathrm{vars}(\psi), \\ (D^\circ, \beta) \models \psi \text{ for all } \psi \in \mathrm{atoms}(v^i), \\ \text{for all } v \in \mathrm{dom}(\beta) \cap \mathrm{dom}(\gamma) \text{ is } \beta(v) = \gamma(v), \\ \text{and } D^\circ_{\mathrm{VAR}}(\beta) \geqslant v^{\lceil\gamma\rceil} \end{array}$$

$$\overset{(*)}{\iff} \begin{array}{c} \beta \supseteq \alpha^i,\ \mathrm{dom}(\beta) = \bigcup_{\psi \in \mathrm{atoms}(v^i)} \mathrm{vars}(\psi), \\ (D^\circ, \beta) \models \psi \text{ for all } \psi \in \mathrm{exatoms}(v^i), \text{ and} \\ \text{for all } j \in [s] \text{ holds } (D^\circ, \beta|_{\mathrm{vpath}[v] \cup \mathsf{succ}(u_j)}) \models \psi \\ \text{for all } \psi \in \mathrm{atoms}(u_j), \\ \text{for all } v \in \mathrm{dom}(\beta) \cap \mathrm{dom}(\gamma) \text{ is } \beta(v) = \gamma(v), \\ \text{and } D^\circ_{\mathrm{VAR}}(\beta) \geqslant v^{\lceil\gamma\rceil} \end{array}$$

$$\iff \begin{array}{c} \beta \supseteq \alpha^i,\ \mathrm{dom}(\beta) = \bigcup_{\psi \in \mathrm{atoms}(v^i)} \mathrm{vars}(\psi), i \text{ is fit in } D^\circ \\ \text{and for all } j \in [s] \text{ there is a } \iota \in \mathrm{Es}^\circ_{u_j}(i, \lceil\gamma\rceil) \\ \text{such that } \beta|_{\mathrm{vpath}[v] \cup \mathsf{succ}(u_j)} \in \left(\Delta^\iota_\gamma\right)^\circ \text{ and} \\ \text{for all } v \in \mathrm{dom}(\beta) \cap \mathrm{dom}(\gamma) \text{ is } \beta(v) = \gamma(v), \text{ and } D^\circ_{\mathrm{VAR}}(\beta) \geqslant v^{\lceil\gamma\rceil} \end{array}$$

The equivalence $(*)$ follows from the fact that

$$\mathrm{atoms}(v^i) = \mathrm{exatoms}(v^i) \cup \mathrm{atoms}(u_1) \cup \cdots \cup \mathrm{atoms}(u_s)\ .$$

By the induction hypothesis it follows that $\beta|_{\mathrm{vpath}[v] \cup \mathsf{succ}(u_j)}$ will be enumerated by DiffEnum$(\iota, \lceil\gamma\rceil, \circ)$. It remains to show the last equivalence, i.e., for all $u \in \mathit{child}(v^i)$ holds $(D^\circ, \beta_u) \models \psi$ for all $\psi \in \mathrm{atoms}(u)$ if and only if for all $u \in \mathit{child}(v^i)$ there is a $\iota \in \mathrm{Es}^\circ_u(i, \lceil\gamma\rceil)$ such that $\beta_u \in \left(\Delta^\iota_\gamma\right)^\circ$ where $\beta_u := \beta|_{\mathrm{vpath}[u] \cup \mathsf{succ}(u)}$.

For the "if" direction let $u \in \mathit{child}(v)$ be arbitrary and assume that $(D^\circ, \beta_u) \models \psi$ for all $\psi \in \mathrm{atoms}(u)$. In the first step we show $\lceil\beta|_{\mathrm{vpath}[u]}\rceil \in \mathrm{Es}^\circ_u(i, \lceil\gamma\rceil)$:

- *Case 1:* $u \in \mathrm{path}[v^{\lceil\gamma\rfloor}]$: Then $\mathrm{Es}^\circ_u(i, \lceil\gamma\rceil) = \left\{\lceil\gamma|_{\mathrm{vpath}[u]}\rceil\right\}$. Since $\beta(v) = \gamma(v)$ for all $v \in \mathrm{dom}(\beta) \cap \mathrm{dom}(\gamma)$ it holds that $\gamma|_{\mathrm{vpath}[u]} = \beta|_{\mathrm{vpath}[u]}$ and therefore $\lceil\beta|_{\mathrm{vpath}[u]}\rceil$ is in $\mathrm{Es}^\circ_u(i, \lceil\gamma\rceil)$.

- *Case 2:* $u \notin \mathrm{path}[v^{\lceil\gamma\rfloor}]$: Since $(D^\circ, \beta_u) \models \psi$ for all $\psi \in \mathrm{atoms}(u)$ it follows that β_u is fit in D° and, in particular, $\lceil\beta|_{\mathrm{vpath}[u]}\rceil$ belongs to $\mathcal{L}_u^{\lceil\beta|_{\mathrm{vpath}[u)}\rceil}$ or $\{\iota \in (\mathcal{L}_u^{\lceil\beta|_{\mathrm{vpath}[u)}\rceil})^\circ \ :\ \mathtt{fit}^\circ[\lceil\beta_u\rceil] \text{ is } \mathsf{true}\}$. Hence, $\lceil\beta|_{\mathrm{vpath}[u]}\rceil \in \mathrm{Es}^\circ_u(i, \lceil\gamma\rceil)$ if $v \notin \mathrm{path}[v^i]$ and $v > D^\circ_{\mathrm{VAR}}(\alpha^{\widehat{\iota}})$. Let us now consider the case that $v < D^\circ_{\mathrm{VAR}}(\alpha^{\widehat{\iota}})$. Assume for a contradiction that $u < D^\circ_{\mathrm{VAR}}(\gamma)$ and $\mathtt{fit}^\circ[\lceil\beta|_{\mathrm{vpath}[u]}\rceil]$ is true. Then it follows that $D^\circ_{\mathrm{VAR}}(\beta_u) \leqslant u$ and thus $D^\circ_{\mathrm{VAR}}(\beta) \leqslant u < v^{\lceil\gamma\rceil}$ (since $\beta|_{\mathrm{vpath}[v]} \subseteq \beta$) and thus $\beta \notin \left(\Delta^i_\gamma\right)^\circ$ which violates $\beta \in \left(\Delta^i_\gamma\right)^\circ$. Hence, $\lceil\beta|_{\mathrm{vpath}[u]}\rceil \in \mathrm{Es}^\circ_u(i, \lceil\gamma\rceil)$ if $v \notin \mathrm{path}[v^i]$ and $v < D^\circ_{\mathrm{VAR}}(\alpha^{\widehat{\iota}})$.

Let $\iota := \lceil \beta|_{\text{vpath}[u]} \rceil$. It remains to show that $\beta_u \in \left(\Delta^{\iota}_{\gamma}\right)^{\circ}$. It is known that $(D^{\circ}, \beta_u) \models \psi$ for all $\psi \in \text{atoms}(v^i)$. Since for all $v \in \text{dom}(\beta) \cap \text{dom}(\gamma)$ we have $\beta(v) = \gamma(v)$ and because $\beta_u \subseteq \beta$ for all $v \in \text{dom}(\beta_u) \cap \text{dom}(\gamma)$ we have $\beta_u(v) = \gamma(v)$.

Assume for a contradiction that $D^{\circ}_{\text{VAR}}(\beta_u) < v^{\lceil \gamma \rceil}$. Then, there is an item i with $\alpha^i \subseteq \beta_u$ with $v^i = D^{\circ}_{\text{VAR}}(\beta_u)$ such that $\texttt{fit}^{\circ}[i]$ it $\texttt{true}$ and $v^i < v^{\lceil \gamma \rceil}$. Since $\beta \supseteq \beta_u$, also $\alpha^i \subseteq \beta$ and therefore $D^{\circ}_{\text{VAR}}(\beta) < v^{\lceil \gamma \rceil}$. Thus, it follows that $\beta \notin \left(\Delta^{i}_{\gamma}\right)^{\circ}$ which violates $\beta \in \left(\Delta^{i}_{\gamma}\right)^{\circ}$.

Therefore $D^{\circ}_{\text{VAR}}(\beta_u) \geqslant v^{\lceil \gamma \rceil}$ and altogether $\beta_u \in \left(\Delta^{\iota}_{\gamma}\right)^{\circ}$.

For the "only if" direction, let us assume that i is fit and for all $u \in \mathit{child}(v^i)$ there is an $\iota \in \text{Es}^{\circ}_u$ such that $\beta_u \in \left(\Delta^{\iota}_{\gamma}\right)^{\circ}$. By the definition of $\left(\Delta^{\iota}_{\gamma}\right)^{\circ}$ and since i is fit in D°, it follows that $(D^{\circ}, \beta) \models \psi$ for all $\psi \in \bigcup_{u \in \mathit{child}(v)} \text{atoms}(u) \cup \text{exatoms}(v) = \text{atoms}(v)$. $\square$

Algorithm 8 does not enumerate a tuple twice since for all ι, ι' that will be yielded by Algorithm 7, it holds $\alpha^{\iota} \not\subseteq \alpha^{\iota'}$ and $\alpha^{\iota'} \not\subseteq \alpha^{\iota}$ and therefore $\left(\Delta^{\iota}_{\gamma}\right)^{\circ} \cap \left(\Delta^{\iota'}_{\gamma}\right)^{\circ} = \emptyset$.

Recall, that we have constructed the data structure such that we can efficiently enumerate the set $\mathcal{L}^i_u \setminus \{\iota : \texttt{fit}^{+}[i] \text{ is } \texttt{true}\}$ and the fact, that the delay takes time $O(\text{vars}(Q)^2)$ can be shown analogously to the delay time of Algorithm 3.

The next claim shows that Algorithm 8 is correct.

Claim 4.30. *The tuples printed in line Algorithm 8 are exactly the tuples that belong to $D^{\circ} \setminus D^{\otimes}$.*

Proof.

$$
\begin{aligned}
&\overline{a} \text{ will be printed in Algorithm 8}\\
\overset{(\star)}{\Longleftrightarrow}\ &\text{there is a } \widehat{\iota} \in \mathsf{EnumFit}(\circ, \texttt{[}\emptyset\texttt{]}) \text{ and a } \alpha \in \mathsf{DiffEnum}(\texttt{[}\emptyset, \widehat{\iota}, \circ\texttt{]})\\
&\qquad \text{such that } a = (\alpha(x_1), \ldots, \alpha(x_\ell), b_1, \ldots, b_\ell)\\
&\\
&\qquad \text{there is an item } \widehat{\iota} = \texttt{[}\beta\texttt{]}\\
\overset{(\star\star)}{\Longleftrightarrow}\ &\text{such that for all } w \in \mathrm{vpath}[v^{\texttt{[}\beta\texttt{]}}] \text{ is } \texttt{[}\beta|_{\mathrm{vpath}[w]}\texttt{]} \text{ fit in } D^\circ \text{ and}\\
&\qquad \texttt{fit}^\circ[\texttt{[}\beta\texttt{]}] \text{ is } \mathsf{true} \text{ and } \texttt{[}\beta\texttt{]} \text{ is not redundant}\\
&\text{and a } \alpha \in \left(\Delta_\beta^{\texttt{[}\emptyset\texttt{]}}\right)^\circ \text{ such that } a = (\alpha(x_1), \ldots, \alpha(x_\ell), b_1, \ldots, b_\ell)\\
&\\
&\qquad \text{there is an item } \widehat{\iota} = \texttt{[}\beta\texttt{]}\\
&\qquad \text{such that for all } w \in \mathrm{vpath}[v^{\texttt{[}\beta\texttt{]}}] \text{ is } \texttt{[}\beta|_{\mathrm{vpath}[w]}\texttt{]} \text{ fit in } D^\circ \text{ and}\\
\overset{(\star\star\star)}{\Longleftrightarrow}\ &\qquad \texttt{fit}^\circ[\texttt{[}\beta\texttt{]}] \text{ is } \mathsf{true} \text{ and } \texttt{[}\beta\texttt{]} \text{ is not redundant}\\
&\qquad \text{and a } \alpha \text{ with } \mathrm{dom}(\alpha) = \mathrm{vars}(Q) \text{ and } (D^\circ, \alpha) \models Q \text{ and}\\
&\beta \subseteq \alpha \text{ and } D^\circ_{\mathrm{VAR}}(\alpha) \geqslant v^{\texttt{[}\beta\texttt{]}} \text{ such that } \overline{a} = (\alpha(x_1), \ldots, \alpha(x_\ell), b_1, \ldots, b_\ell)\\
&\\
&\qquad \text{there is an assigment } \alpha \text{ with } \mathrm{dom}(\alpha) = \mathrm{vars}(Q) \text{ and}\\
\overset{(\star\star\star\star)}{\Longleftrightarrow}\ &\overline{a} = (\alpha(x_1), \ldots, \alpha(x_\ell), b_1, \ldots, b_\ell) \text{ and there is an item } \widehat{\iota} = \texttt{[}\beta\texttt{]}\\
&\qquad \text{with } \beta \subseteq \alpha \text{ such that for all } w \in \mathrm{free}(Q) \text{ we have}\\
&\qquad \texttt{[}\alpha|_{\mathrm{vpath}[w]}\texttt{]} \text{ is fit in } D^\circ, \text{and } \texttt{fit}^\circ[\widehat{\iota}] \text{ is } \mathsf{true}\\
&\\
\overset{4.18}{\Longleftrightarrow}\ &\overline{a} = (\alpha(x_1), \ldots, \alpha(x_k), b_1, \ldots, b_\ell) \in Q(D^\circ) \setminus Q(D^\otimes)
\end{aligned}
$$

Equation $(\star)$ follows from the construction of the algorithm, Equation $(\star\star)$ by Claim 4.24 and 4.29, Equation $(\star\star\star)$ by the Definition of $\left(\Delta_\beta^{\texttt{[}\emptyset\texttt{]}}\right)^\circ$. For the "if" direction in $(\star\star\star\star)$ Note that for all $w \in \mathrm{vars}(Q)$ we have $\texttt{[}\alpha|_{\mathrm{vpath}[w]}\texttt{]}$ is fit in D° follows from $(D^\circ, \alpha) \models Q$. For the "only if" direction of $(\star\star\star\star)$ note that one can choose an item $\widehat{\iota}$ with $D^\circ_{\mathrm{VAR}}(\alpha) = v^{\widehat{\iota}}$ and by Definition of $D^\circ_{\mathrm{VAR}}(\alpha)$ and the fact that the variables are sorted in pre-order, it follows that $\widehat{\iota}$ is not $\circ$-redundant. □

This concludes the proof of Theorem 3.3(c).

4.8. Outputting the next smallest tuple

In this section we will show that for a lexicographically ordered data structure, we can do the following routine. Upon input of a tuple $\overline{a} \in \mathbf{dom}^{k+\ell}$, output the following tuple

$$\max\{\overline{c} : \overline{c} \leqslant_{\mathrm{lex}} \overline{a},\ \overline{c} \in Q(D)\}$$

if it exists, and `SmallerThanMinimum` otherwise.

The result is stated in the following lemma:

Lemma 4.31. *There is a dynamic algorithm that receives a q-hierarchical k-ary CQ Q and a σ-db D_0 of size $\|D_0\|$, and computes within preprocessing time $\mathit{poly}(Q) \cdot O(\|D_0\| \log(\|D_0\|))$ a data structure that can be updated in time $\mathit{poly}(Q) \cdot O(\log(\|D\|))$ and allows the following:*

(a) enumerate the tuples in $Q(D)$ in lexicographical order with delay $\mathit{poly}(Q)$ and

(b) upon input of a tuple $\overline{b} \in \mathbf{dom}^{k+\ell}$ output the tuple

$$\max\left\{\overline{a} \in Q(D) \ : \ \overline{a} \leqslant \overline{b}\right\}$$

if it exists, or a `SmallerThanMinimum`-message otherwise in time $t_l = \mathit{poly}(Q)$

where D is the current database.

We have already seen that updating a lexicographically ordered data structure can be done in $O(\log \|D\|)$ and it supports enumerating the query result in lexicographical order (see Remark 4.9 and 4.15). In particular, this data structure will be used for Lemma 4.31. Moreover, the routine in Lemma 4.31 (a) is also supported. The aim of this section is to show that Lemma 4.31 (b) holds. Before we discuss the general case, let us consider a running example.

Example 4.32. *Suppose, we have the lexicographically ordered data structure for Q on D_0 from Example 4.4. Note that we obtain the data structure from Figure 4.3 where the items $\llbracket \frac{2\ 2\ 8}{yx_2x_3} \rrbracket$ and $\llbracket \frac{2\ 2\ 4}{yx_2x_3} \rrbracket$ are inverted in the x_3-list of $\llbracket \frac{2\ 2\ 8}{yx_2x_3} \rrbracket$.*

Assume, we receive as input the tuple $\overline{a} = (1, 2, 3, 1)$ and the aim is to compute the maximum tuple $\overline{c}$ that belongs to the query result and is smaller than $\overline{a}$, i.e., the tuple $\overline{c} = (1, 1, 6, 4)$. Let α be the assignment that coincides with $\overline{a}$, i.e., $\alpha = \frac{1\ 2\ 3\ 1}{yx_1x_2x_3}$. In the first step, we compute two variables $\widetilde{v}$ and v. The variable $\widetilde{v} \in \{y, x_1, x_2, x_3\}$ is the minimal variable such that $\llbracket \alpha|_{\mathrm{vpath}[\widetilde{v}]} \rrbracket$ is not in the $\widetilde{v}$-list of $\llbracket \alpha|_{\mathrm{vpath}[\widetilde{v})} \rrbracket$. In our example, this is $\widetilde{v} = x_2$. Then, we let $v \in \{y, x_1, x_2\} = \{w \in \mathrm{vars}(Q) : w \leqslant x_2\}$ be the maximum variable such that there is an item $\iota \in \mathcal{L}_v^{\llbracket \alpha|_{\mathrm{vpath}[v]} \rrbracket}$ with $a^\iota < \alpha(v)$, and v is smaller than or equal to $\widetilde{v}$. In our example, $v = x_1$, since there is an item $\llbracket \frac{1\ 1}{yx_1} \rrbracket$ that appears before $\llbracket \frac{1\ 2}{yx_1} \rrbracket$ in the x_1-list of $\llbracket \frac{1}{y} \rrbracket$.

The idea is the following. Since the lists in the data structure are sorted, the enumeration algorithm in Algorithm 2 will enumerate the tuples in lexicographical order. Therefore, we ask for the maximum tuple, the enumeration algorithm will output, that is smaller than $\overline{a}$. Note that we have selected $\widetilde{v}$ as the minimum variable, such that $\llbracket \alpha|_{\mathrm{vpath}[\widetilde{v}]} \rrbracket$ is not in the $x_{\widetilde{v}}$-list of $\llbracket \alpha|_{\mathrm{vpath}[\widetilde{v})} \rrbracket$. In particular, it follows that in the for-loop in line 4 we will not iterate $\llbracket \frac{1\ 3}{yx_2} \rrbracket$. Since this is the minimum variable for which the condition holds, the for-loop in line 4 is the outermost for-loop where we do not find a corresponding item of the form $\llbracket \alpha|_{\mathrm{path}[u]} \rrbracket$. Therefore we upwardly try to find upwardly a for-loop that has a previous step for $\llbracket \frac{1\ 3}{yx_2} \rrbracket$. This is not in line 4 the case since the first element in the x_2-list of $\llbracket \frac{1}{y} \rrbracket$ is $\llbracket \frac{1\ 4}{yx_2} \rrbracket$. But the for-loop in line 3 has a previous element for $\llbracket \frac{1\ 2}{yx_1} \rrbracket$ in the x_1-list of $\llbracket \frac{1}{y} \rrbracket$. This is exactly the condition that holds for v. Let us assume we set the for-loop in line 3 to the previous step, the set

the for-loop in line 2 to the configuration that matches with $\overline{a}$ and the for-loop in line 4 and 5 to the last configuration. Then, the tuple is smaller then $\overline{a}$ and its successor tuple in the enumeration is lexicographically greater.

In other words, we construct the following assignment.

- *For all $v' < v$, we set $\gamma(v') = \alpha(v)$. Then, $\gamma(y) = 1$.*
- *Then, we set $\gamma(v)$ to the constant of the item that appears before $\lceil \frac{1\ 2}{yx_1} \rceil$, i.e., $\gamma(x_1) = 1$.*
- *For all $v' > v$, we set $\gamma(v') = a^\iota$, where ι is the last element of x_j-list of $\lceil \gamma|_{\mathrm{vpath}[x_j)} \rceil$, i.e., $\gamma(x_2) = 6$ and $\gamma(x_3) = 4$*

We obtain $\overline{c}$ by setting $\overline{c} = (\gamma(y), \gamma(x_1), \gamma(x_2), \gamma(x_3)) = (1, 1, 6, 4)$. If $\overline{a}$ would appear in the enumeration algorithm, it would be followed by $\overline{c}$.

Now we consider the general case. In the reminder of this section we will show that Lemma 4.31 (b) holds.

For the remainder of this proof we assume that $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is a q-hierarchical conjunctive query, $\mathrm{vars}(Q) = \{x_1, \ldots, x_m\}$ with $0 \leqslant k \leqslant m$, and Q is of the form

$$Q \quad = \quad \left\{(x_1 \ldots x_k, b_1, \cdots, b_\ell) \ : \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right)\right\}, \tag{4.9}$$

where $b_1, \ldots, b_\ell \in \mathbf{dom}$ and $\psi_1, \ldots, \psi_d$ are atomic queries of schema σ. Let

$$\varphi \ := \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right).$$

Let $\overline{a}$ be the tuple we receive as input. The algorithm for computing the tuple

$$\overline{c} \ = \ \max\left\{\overline{d} \in Q(D) \ : \ \overline{d} \leqslant \overline{a}\right\}$$

is the following. If $\overline{a}$ is smaller than the minimum element, we output the message `SmallerThanMinimum` and stop the algorithm. If $\overline{a} \in Q(D)$ we output $\overline{a}$ and stop the algorithm. Let α be the assignment such that $\alpha(x_j) := a_j$ for all $j \in [k]$. If $\overline{a} \notin Q(D)$ and $(D, \alpha) \not\models \varphi$ we construct an assignment γ such that for the number

$$p \ := \ \max\left\{j \in [\hat{p}] \ : \text{ there is an } \iota \in \mathcal{L}_{x_j}^{\lceil \alpha|_{\mathrm{vpath}[x_j)} \rceil} \text{ such that } a^\iota < a_i\right\}$$

where

$$\hat{p} := \min\left\{j \in [k] : \lceil \alpha|_{\mathrm{vpath}[x_j]} \rceil \text{ is not in the } x_j\text{-list of } \lceil \alpha|_{\mathrm{vpath}[x_j)} \rceil\right\}$$

the following holds:

- for all $j < p$ is $\gamma(x_j) = \alpha(x_j)$ and
- $\gamma(x_p) = \max\left\{a^\iota \ : \ \iota \in \mathcal{L}_{x_\ell}^{\lceil \alpha|_{\mathrm{vpath}[x_\ell)} \rceil},\ a^\iota < a_i\right\}$ and

- for all $j > p$ is $\gamma(x_j) = \alpha^\iota$ the maximum item ι of the x_j-list of $\texttt{[}\gamma|_{\text{vpath}[x_j)}\texttt{]}$.

Note that an index p exists since $\overline{a}$ is not smaller than the minimum. As a remark, note that x_p coincides with v in Example 4.32 and $x_{\widetilde{p}}$ with $\widetilde{v}$. Note that the elements can be computed if γ will be constructed by traversing through the nodes of the query tree using the pre-order method. The algorithm outputs $\overline{c} = (\gamma(x_1), \ldots, \gamma(x_k), b_1, \ldots, b_\ell)$. Note that $\overline{c} < \overline{a}$ since by construction of γ follows that $(\gamma(x_1), \ldots, \gamma(x_k)) < (a_1, \ldots, a_k)$.

Let us consider the case $\overline{a} \notin Q(D)$ and $(D, \alpha) \models \varphi$. If $(a_{k+1}, \ldots, a_{k+\ell}) > (b_1, \ldots, b_\ell)$, output the tuple $\overline{c} = (a_1, \ldots, a_k, b_1, \ldots, b_\ell)$. It is straightforward to see that $\overline{c}$ is the sought tuple. If $(a_{k+1}, \ldots, a_{k+\ell}) < (b_1, \ldots, b_\ell)$, use the algorithm for the case that $\overline{a} \notin Q(D)$ and $(D, \alpha) \not\models \varphi$, but the number $\hat{p}$ is the number k (instead of setting $\hat{p}$ to the minimum index such that $\texttt{[}\alpha|_{\text{vpath}[x_p]}\texttt{]}$ is not in the x_p-list of $\texttt{[}\alpha|_{\text{vpath}[x_p)}\texttt{]}$).

We now show that the choice of the tuple above is correct and that the tuple can be computed in time $O(\mathit{poly}(Q)\|D\|)$.

Correctness It is easy to see that the algorithm is correct if $\overline{a} \in Q(D)$ or $\overline{a}$ is smaller than the minimum tuple in $Q(D)$. The case that $\overline{a} \notin Q(D)$ and $(D, \alpha) \models \varphi$ and $(a_{k+1}, \ldots, a_{k+\ell}) < (b_1, \ldots, b_\ell)$ is straightforward.

Let us consider the other cases: Let us first look upon the case that $\overline{a} \notin Q(D)$ and $(D, \alpha) \not\models \varphi$. The constructed tuple $\overline{c}$ belongs to the query result since the items $\texttt{[}\gamma|_{\text{vpath}[v]}\texttt{]}$ are fit for all $v \in \text{vars}(Q)$. Let us assume for a contradiction that there is a tuple $\overline{d} \in Q(D)$ with $\overline{c} <_{\text{lex}} \overline{d} <_{\text{lex}} \overline{a}$ where $\overline{c} := (\gamma(x_1), \ldots, \gamma(x_p), b_1, \ldots, b_\ell)$. Then, there is a $p' \in \{p, \ldots, k+\ell\}$ such that for all $j < p'$ is $c_j = d_j$ and $c_{p'} < d_{p'} \leqslant a_{p'}$. (Here p is defined as in the algorithm above) Note that by construction is $c_j = a_j$ for all $j < p$ and, in particular, we have $c_j = d_j = a_j$ for all $j < p$.

- *Case 1: $p' = p$ and $d_{p'} < a_{p'}$*
  In that case, there is an item $\texttt{[}\alpha|_{\text{vpath}[x_p)} \cup \{x_p \mapsto d_p\}\texttt{]}$ in the x_p-list of $\texttt{[}\alpha|_{\text{vpath}[x_p)}\texttt{]}$ with $c_p < d_p < a_p$. This violates the choice of c_p.

- *Case 2: $p' = p$ and $d_p = a_p$*
  Since $\overline{d} <_{\text{lex}} \overline{a}$ there is a $\tilde{p}$ such that for all $j < \tilde{p}$ it holds that $d_j = a_j$ and $d_{\tilde{p}} < a_{\tilde{p}}$. Because $\overline{d} \in Q(D)$ it follows that $\texttt{[}\alpha|_{\text{vpath}[x_{\tilde{p}})} \cup \left\{x_{\widetilde{p}} \mapsto d_{\tilde{p}}\right\}\texttt{]}$ is in the $x_{\widetilde{p}}$-list of $\texttt{[}\alpha|_{\text{vpath}[x_{\tilde{p}})}\texttt{]}$.

  Because $\overline{d} \in Q(D)$ and since $\texttt{[}\alpha|_{\text{vpath}[x_{\hat{p}}]}\texttt{]}$ is not in the $x_{\hat{p}}$-list of $\texttt{[}\alpha|_{\text{vpath}[x_{\hat{p}})}\texttt{]}$ it holds that $\tilde{p} \leqslant \hat{p}$ as $d_p \neq a_p$ (Note that otherwise, the item $\texttt{[}\alpha|_{\text{vpath}[x_{\hat{p}}]}\texttt{]}$ must be fit)..

  All in all, we have that there is an index $\widetilde{p} \in \{p+1, \ldots, \widehat{p}\}$ such that there is an item ι in the $x_{\widetilde{p}}$-list of $\texttt{[}\alpha|_{\text{vpath}[x_{\tilde{p}})}\texttt{]}$ with $\alpha^\iota < a_{\widetilde{p}}$. This is a contradiction to the choice of p.

- *Case 3: $p' > p$*
  Then, there is an item $\texttt{[}\gamma|_{\text{vpath}[x_{p'})} \cup \{x_{p'} \mapsto d_{p'}\}\texttt{]}$ in the $x_{p'}$-list of $\texttt{[}\gamma|_{\text{vpath}[x_{p'})}\texttt{]}$ with $c_{p'} < d_{p'}$. But this is a contradiction to the fact that $c_{p'}$ is the maximum element of the x_j-list of $\texttt{[}\gamma|_{\text{vpath}[x_j)}\texttt{]}$.

The proof for the case that $\overline{a} \notin Q(D)$ and $(D, \alpha) \models \varphi$ and $(a_{k+1}, \ldots, a_{k+\ell}) < (b_1, \ldots, b_\ell)$ can be taken verbatim from the case that $\overline{a} \notin Q(D)$ and $(D, \alpha) \not\models \varphi$, by removing the sentence " Because $\overline{d} \in Q(D)$ and since $\mathtt{[}\alpha|_{\mathrm{vpath}[x_{\hat{p}}]}\mathtt{]}$ is not in the $x_{\hat{p}}$-list of $\mathtt{[}\alpha|_{\mathrm{vpath}[x_{\hat{p}}]}\mathtt{]}$ it holds that $\tilde{p} \leqslant \hat{p}$ as $d_p \neq a_p$ (Note that otherwise, the item $\mathtt{[}\alpha|_{\mathrm{vpath}[x_{\hat{p}}]}\mathtt{]}$ must be fit).". Note that the fact $\widetilde{p} \leqslant \widehat{p}$ is clear since $\widehat{p} = k$.

Running time First, we compute the minimum element of $Q(D)$ by starting the enumeration procedure until the first tuple will be output and test if $\overline{a}$ is smaller than the minimum element. This can be done in $\mathit{poly}(Q)$. Then, we test in time $\mathit{poly}(Q)$ if $\overline{a} \in Q(D)$ and output in that case $\overline{a}$. In order to compute the number $\hat{p}$, it takes time $\mathit{poly}(Q)$ to test if $\mathtt{[}\alpha|_{\mathrm{vpath}[x_j]}\mathtt{]}$ is in the x_j-list of $\mathtt{[}\alpha|_{\mathrm{vpath}[x_j)}\mathtt{]}$, i.e., to test if $\mathtt{[}\alpha|_{\mathrm{vpath}[x_j]}\mathtt{]}$ is fit. Then, we find in time $\mathit{poly}(Q)$ the maximum index p such that $\tilde{p} \leqslant \hat{p}$ and $\mathtt{[}\alpha|_{\mathrm{vpath}[x_{\tilde{p}}]}\mathtt{]}$ is not the first item in the $x_{\tilde{p}}$- list of $\mathtt{[}\alpha|_{\mathrm{vpath}[x_{\tilde{p}})}\mathtt{]}$. To compute the element c_p, it takes time $\mathit{poly}(Q)\log(\|D\|)$ to find the element using the binary search method if $p = \widehat{p}$. Otherwise, one can simply take the previous item of the item ι with $a^\iota = \alpha(x_p)$. The other components of $\overline{c}$ can be computed in constant time. Thus, it takes time $\mathit{poly}(Q)\log(\|D\|)$ to compute the tuple.

This concludes the proof of Lemma 4.31 (b).

5. Testing t-hierarchical conjunctive queries under updates

The aim of this section is to give a precise characterisation of the CQs for which *testing* can be done efficiently under updates. In this chapter, a proof for Theorem 3.6 is given.

Theorem 3.6 is restated in the following.

Theorem 3.6 ([21]). *There is a dynamic algorithm that receives a t-hierarchical k-ary CQ Q and a σ-db D_0, and computes within $t_p = poly(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows to test for an input tuple $\overline{a} \in \mathbf{dom}^k$ if $\overline{a} \in Q(D)$ within time $t_t = poly(Q)$, where D is the current database.*

The remainder of this chapter is based on [21].

Of course, according to Theorem 3.3 (a), the testing problem can be solved with constant update time and constant testing time for every *q-hierarchical* CQ. But the same holds true for the non-q-hierarchical CQ $Q_{S\text{-}E\text{-}T} := \{ (x, y) : Sx \wedge Exy \wedge Ty \}$. The corresponding dynamic algorithm simply uses 1-dimensional arrays $\mathtt{A}_S$ and $\mathtt{A}_T$ and a 2-dimensional array $\mathtt{A}_E$ such that for all $a, b \in \mathbf{dom}$ we have $\mathtt{A}_E[a, b] = 1$ if $(a, b) \in E^D$, and $\mathtt{A}_E[a, b] = 0$ otherwise, and $\mathtt{A}_R[a] = 1$ if $a \in R^D$, and $\mathtt{A}_R[a] = 0$ otherwise, for $R \in \{S, T\}$. When given an update command, the arrays can be updated within constant time. And when given a tuple $(a, b) \in \mathbf{dom}^2$, the **test** routine simply looks up the array entries $\mathtt{A}_S[a]$, $\mathtt{A}_E[a, b]$, $\mathtt{A}_T[b]$ and returns the correct query result accordingly. To characterise the conjunctive queries for which testing can be done efficiently under updates, we consider *t-hierarchical* CQs (see Definition 3.4).

Obviously, it can be checked in time $poly(Q)$ whether a given CQ Q is t-hierarchical. Note that every q-hierarchical CQ is t-hierarchical, and a *Boolean* query is t-hierarchical if and only if it is q-hierarchical. The aim of this chapter is to show Theorem 3.6, i.e., to show that the *t-hierarchical* CQs characterise CQs for which the *testing* problem can be solved efficiently under updates. In the remainder of this section, we give a proof for Theorem 3.6.

To avoid notational clutter, and without loss of generality, we restrict attention to queries $Q_\varphi(u_1, \ldots, u_k, b_1, \ldots, b_\ell)$ where $(u_1, \ldots, u_k)$ is of the form $(z_1, \ldots, z_k)$ for pairwise distinct variables $z_1, \ldots, z_k$ and $b_1, \ldots, b_\ell \in \mathbf{dom}$. We combine the array construction described above for the example query $Q_{S\text{-}E\text{-}T}$ with the dynamic algorithm provided by Theorem 3.3 (a) and the following Lemma 5.1. To formulate the lemma, we need the following notation. A k-ary *generalised CQ* is of the form $\{ (z_1, \ldots, z_k, b_1, \ldots, b_\ell) : \varphi_1 \wedge \cdots \wedge \varphi_m \}$ where $k \geqslant 0$, $z_1, \ldots, z_k$ are pairwise distinct variables, $m \geqslant 1$, φ_j is a conjunctive formula for each $j \in [m]$, $\mathrm{free}(\varphi_1) \cup \cdots \cup \mathrm{free}(\varphi_m) =$

$\{z_1, \ldots, z_k\}$, and the quantified variables of φ_j and $\varphi_{j'}$ are pairwise disjoint for all $j, j' \in [m]$ with $j \neq j'$ and disjoint from $\{z_1, \ldots, z_k\}$. For each $j \in [m]$ let $\overline{z}^{(j)}$ be the sublist of $\overline{z} := (z_1, \ldots, z_k)$ that only contains the variables in free(φ_j) and let $\overline{b}^{(j)}$ be the sublist of $\overline{b} := (b_1, \ldots, b_\ell)$ that only contains the constant in φ_j. I.e., $\overline{z}^{(j)}$ is obtained from $\overline{z}$ by deleting all variables that do not belong to free(φ_j) and $\overline{b}^{(j)}$ is obtained from $\overline{b}$ by deleting all constants that do not appear in φ_j. Accordingly, for a tuple $\overline{a} = (a_1, \ldots, a_k) \in \mathbf{dom}^k$ by $\overline{a}^{(j)}$ we denote the tuple that contains exactly those a_i where z_i belongs to $\overline{z}^{(j)}$. The query result of Q on a σ-db D is the set

$$Q(D) \;:=\; \left\{ \overline{a}, \overline{b} \in \mathbf{dom}^k \;:\; D \models \varphi_j[\overline{a}^{(j)}] \;\text{ for each } j \in [m] \right\},$$

where $D \models \varphi_j[\overline{a}^{(j)}]$ means that there is a homomorphism $\beta_j : Q_j \to D$ for the query $Q_j := \{\, \overline{z}^{(j)}, \overline{b}^{(j)} \;:\; \varphi_j \,\}$, with $\beta_j(z_i) = a_i$ for every i with $z_i \in \text{free}(\varphi_j)$ and $\beta_j(c) = c$ for all $c \in \text{cons}(\varphi_j)$. For example,

$$Q'_{E\text{-}E\text{-}R} := \{\, (x, y) \;:\; \exists v_1\, Exv_1 \;\wedge\; \exists v_2\, Eyv_2 \;\wedge\; \exists v_3\, Rxyv_3 \,\}$$

is a generalised CQ that is equivalent to the CQ $Q_{E\text{-}E\text{-}R}$. The proof of the following lemma is given at the end of the section.

Lemma 5.1. *Every t-hierarchical CQ $Q_\varphi(z_1, \ldots, z_k, b_1, \ldots, b_\ell)$ is equivalent to a generalised CQ $Q' = \{\, (z_1, \ldots, z_k, b_1, \ldots, b_\ell) : \varphi_1 \wedge \cdots \wedge \varphi_m \,\}$ such that for each $j \in [m]$ the CQ $Q_j := \{\, \overline{z}^{(j)} : \varphi_j \,\}$ is q-hierarchical or quantifier-free. Furthermore, there is an algorithm which decides in time poly(Q_φ) whether Q_φ is t-hierarchical, and if so, outputs an according Q'.*

When given a t-hierarchical CQ $Q_\varphi(z_1, \ldots, z_k)$, use the algorithm provided by Lemma 5.1 to compute an equivalent generalised CQ Q' of the form $\{(z_1, \ldots, z_k) : \varphi_1 \wedge \cdots \wedge \varphi_m\}$ and let $Q_j := \{\overline{z}^{(j)}, \overline{b}^{(j)} \;:\; \varphi_j\}$ for each $j \in [m]$. W.l.o.g. assume that there is a $m' \in \{0, \ldots, m\}$ such that Q_j is q-hierarchical for each $j \leqslant m'$ and Q_j is quantifier-free for each $j > m'$. We use in parallel, for each $j \leqslant m'$, the data structures provided by Theorem 3.3 (a) for the q-hierarchical CQ Q_j. In addition to this, we use an r-dimensional array $\mathtt{A}_R$ for each relation symbol $R \in \sigma$ of arity $r := \text{ar}(R)$, and we ensure that for all $\overline{c} \in \mathbf{dom}^r$ we have $\mathtt{A}_R[\overline{c}] = 1$ if $\overline{c} \in R^D$, and $\mathtt{A}_R[\overline{c}] = 0$ otherwise. When receiving an update command $\mathsf{update}\, R(\overline{c})$, we let $\mathtt{A}_R[\overline{c}] := 1$ if $\mathsf{update} = \mathsf{insert}$, and $\mathtt{A}_R[\overline{c}] := 0$ if $\mathsf{update} = \mathsf{delete}$, and in addition to this, we call the **update** routines of the data structure for Q_j for each $j \leqslant m'$. Upon input of a tuple $(\overline{a}, \overline{d}) \in \mathbf{dom}^{k+\ell}$, the **test** routine proceeds as follows. We test if $\overline{d} = \overline{b}$. If the test fails, we terminate the routine and output `False`. Otherwise we continue as follows. For each $j \leqslant m'$, it calls the **test** routine of the data structure for Q_j upon input $\overline{a}^{(j)}$. Additionally, it uses the arrays $\mathtt{A}_R$ for all $R \in \sigma$ to check if for each $j > m'$ the quantifier-free query Q_j is satisfied by the tuple $\overline{a}^{(j)}$. All this is done within time *poly*(q), and we know that $\overline{a} \in Q(D)$ if and only if all these tests succeed. This completes the proof of Theorem 3.6.

For the remainder of this section we give a proof for Lemma 5.1.

Proof of Lemma 5.1. Along Definition 3.4 it is straightforward to construct an algorithm that decides in time *poly*(Q) whether a given CQ Q is t-hierarchical.

Let $Q := Q_\varphi(z_1, \ldots, z_k, b_1, \ldots, b_\ell)$ be a given t-hierarchical CQ. Let A_0 be the set of all atoms ψ of Q with $\text{vars}(\psi) \subseteq \text{free}(Q)$, and let φ_0 be the quantifier-free conjunctive formula

$$\varphi_0 \;:=\; \bigwedge_{\psi \in A_0} \psi\,.$$

For each $Z \subseteq \text{free}(Q)$ let A_Z be the set of all atoms ψ of Q such that $Z = \text{vars}(\psi) \cap \text{free}(Q)$ and $\text{vars}(\psi) \supsetneq Z$. Let $Z_1, \ldots, Z_n$ (for $n \geqslant 0$) be a list of all those $Z \subseteq \text{free}(Q)$ with $A_Z \neq \emptyset$. For each $j \in [n]$ let $A_j := A_{Z_j}$ and let $Y_j := \big(\bigcup_{\psi \in A_j} \text{vars}(\psi)\big) \setminus Z_j$.

Claim 5.2. *$Y_j \cap Y_{j'} = \emptyset$ for all $j, j' \in [n]$ with $j \neq j'$.*

Proof. We know that $Z_j \neq Z_{j'}$. W.l.o.g. there is a $z \in Z_j$ with $z \notin Z_{j'}$.

For contradiction, assume that $Y_j \cap Y_{j'}$ contains some variable y. Then, $y \in \text{vars}(\psi)$ for some $\psi \in A_j$ and $y \in \text{vars}(\psi')$ for some $\psi' \in A_{j'}$. By definition of A_j we know that $\text{vars}(\psi) \cap \text{free}(Q) = Z_j$, and hence $z \in \text{vars}(\psi)$. By definition of $A_{j'}$ we know that $\text{vars}(\psi') \cap \text{free}(Q) = Z_{j'}$, and hence $z \notin \text{vars}(\psi')$. Hence, $\psi \in \text{atoms}(z)$ and $\psi' \notin \text{atoms}(z)$. Since $\psi \in \text{atoms}(y)$ and $\psi' \in \text{atoms}(y)$, we obtain that $\text{atoms}(z) \cap \text{atoms}(y) \neq \emptyset$ and $\text{atoms}(y) \not\subseteq \text{atoms}(z)$. But by assumption, Q is t-hierarchical, and this contradicts condition (ii) of Definition 3.4. □

For each $j \in [n]$ consider the conjunctive formula

$$\varphi_j \;:=\; \exists y_1^{(j)} \cdots \exists y_{\ell_j}^{(j)} \bigwedge_{\psi \in A_j} \psi\,,$$

where $\ell_j := |Y_j|$ and $(y_1^{(j)}, \ldots, y_{\ell_j}^{(j)})$ is a list of all variables in Y_j. Using Claim 5.2, it is straightforward to see that

$$Q' \;:=\; \{\; (z_1, \ldots, z_k, b_1, \ldots, b_\ell) \;:\; \varphi_0 \wedge \bigwedge_{j \in [n]} \varphi_j \;\}$$

is a generalised CQ that is equivalent to Q. Furthermore, Q' can be constructed in time *poly*(Q). To complete the proof of Lemma 5.1 we consider for each $j \in [n]$ the CQ

$$Q_j \;:=\; \{\; \overline{z}^{(j)} \;:\; \varphi_j \;\}\,,$$

where $\overline{z}^{(j)}$ is a tuple of length $|Z_j|$ consisting of all the variables in Z_j.

Claim 5.3. *Q_j is q-hierarchical, for each $j \in [n]$.*

Proof. First of all, note that Q_j satisfies condition (ii) of Definition 3.1, since $\text{free}(Q_j) = Z_j$, $\text{atoms}_{Q_j}(z) = A_j$ for every $z \in Z_j$, and $\text{atoms}_{Q_j}(y) \subseteq A_j$ for every $y \in Y_j = \text{vars}(Q_j) \setminus \text{free}(Q_j)$.

For contradiction, assume that Q_j is not q-hierarchical. Then, Q_j violates condition (ii) of Definition 3.1. I.e., there are variables $x_1, x_2 \in Z_j \cup Y_j$ and atoms

$\psi_1, \psi_2, \psi_{12} \in A_j$ such that $\text{vars}(\psi_1) \cap \{x_1, x_2\} = \{x_1\}$, $\text{vars}(\psi_2) \cap \{x_1, x_2\} = \{x_2\}$, and $\text{vars}(\psi_{12}) \cap \{x_1, x_2\} = \{x_1, x_2\}$. Since $\text{vars}(\psi) \cap \text{free}(Q) = Z_j$ for all $\psi \in A_j$, we know that $x_1, x_2 \notin \text{free}(Q)$. Therefore, $x, x' \in \text{vars}(q) \setminus \text{free}(Q)$, and hence ψ_1, ψ_2, ψ_{12} are atoms of Q which witness that condition (i) of Definition 3.4 is violated. This contradicts the assumption that Q is t-hierarchical. □

This completes the proof of Lemma 5.1. □

6. Answering q-hierarchical conjunctive queries with aggregates under updates

This chapter is devoted to q-hierarchical conjunctive queries with aggregates. First of all, we provide an example that motivates queries with aggregates. In this chapter, Theorem 3.7, that is restated in the following, will be proven.

Theorem 3.7. *There is a dynamic algorithm that receives a q-hierarchical CQ with aggregates Q with aggregation time t_a and a σ-db D_0, and computes within $t_i = O(1)$ initialisation time and $t_p = poly(Q)t_a O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)t_a$ and allows to*

(a) enumerate $Q(D)$ with delay $t_d = poly(Q)$,

(b) test for an input tuple $\overline{a} \in \mathbf{dom}^k$ if $\overline{a} \in Q(D)$ within time $t_t = poly(Q)$,

(c) enumerate the tuples that join and omit the result set $Q(D)$ after an arbitrary number of updates on a previous chosen state the database received with delay $t_{di} = poly(Q)$ and

(d) compute the cardinality $|Q(D)|$ in time $t_c = O(1)$

where D is the current database.

6.1. Examples for queries with aggregates

Example 6.1. *Let us assume we have a database D that consists of two tables Salary^D and Person^D. In Salary^D we have triples $(a_{\mathsf{pid}}, a_{\mathsf{project}}, a_{\mathsf{salary}})$ where a_{pid} is a personal id, a_{project} is the name of a project and a_{salary} is the salary that the person with id a_{pid} receives for project a_{project}. Here, we assume that a person might be involved in multiple projects. In Person^D we have pairs $(a_{\mathsf{pid}}, a_{\mathsf{name}})$ where a_{pid} is a personal id and a_{name} is the name of the person with id a_{pid}. Let us consider the following query:*

$$Q = \left\{ (x_{pid}, x_{name}, x_{project}, x_{salary}) \ : \ \begin{array}{c} \mathsf{Person}(x_{pid}, x_{name}) \wedge \\ \mathsf{Salary}(x_{pid}, x_{project}, x_{salary}) \end{array} \right\}$$

This query describes tuples $(a_{pid}, a_{name}, a_{project}, a_{salary})$ where a_{pid} is the id for a_{name} who receives a salary of a_{salary} for project $a_{project}$. In the following, some interesting database queries are given.

- *Output name and id of the people and the sum of their salaries.*
- *Output the average salary of each individual project.*
- *Output the project with highest total salary.*

To treat such queries we now introduce queries with aggregates. Later, we will present queries for the statements in Example 6.1.

First, we provide a formal definition for aggregate functions that is similar to the definition fo aggregate function in [73]. For every set X and every $m \in \mathbb{N}_{\geqslant 0}$ and every tuple $(x_1, \ldots, x_m) \in X^m$ let $\{\!\{x_1, \ldots, x_m\}\!\}$ be the multiset with the elements $x_1, \ldots, x_m$. The number m is the cardinality, denoted by $|\{\!\{x_1, \ldots, x_m\}\!\}|$. With $\{\!\{X\}\!\}_m$ we denote the set of the multisets over X with cardinality m. For two multisets $\{\!\{x_1, \ldots, x_m\}\!\}$ and $\{\!\{y_1, \ldots, y_{m'}\}\!\}$ we define their union as

$$\{\!\{x_1, \ldots, x_m\}\!\} \cup \{\!\{y_1, \ldots, y_{m'}\}\!\} := \{\!\{x_1, \ldots, x_m, y_1, \ldots, y_{m'}\}\!\}.$$

An *aggregate function* $\mathcal{F}$ is a class of functions $\mathcal{F} = \{f_n\}_{n \in \mathbb{N}}$ such that $f_n : \{\!\{X\}\!\}_n \to X$.

Example 6.2. *Examples for aggregate functions are:*

- $\mathsf{sum} = \{s_n\}_{n \in \mathbb{N}}$ *with* $X = \mathbb{N}$ *and* $s_0 := 0$*, and* $s_n(\{\!\{x_1, \ldots, x_n\}\!\}) := x_1 + \cdots + x_n$ *for all* $n \in \mathbb{N}_{\geqslant 1}$*,*
- $\mathsf{prod} = \{p_n\}_{n \in \mathbb{N}}$ *with* $X = \mathbb{N}$ *and* $p_0 := 1$*, and* $p_n(\{\!\{x_1, \ldots, x_n\}\!\}) := x_1 \cdot \cdots \cdot x_n$ *for all* $n \in \mathbb{N}_{\geqslant 1}$*,*
- $\mathsf{count} = \{c_n\}_{n \in \mathbb{N}}$ *with* $X = \mathbb{R}$ *and* $c_0 := 0$*, and* $c_n(\{\!\{x_1, \ldots, x_n\}\!\}) := n$ *for all* $n \in \mathbb{N}_{\geqslant 1}$*,*
- $\mathsf{avg} = \{a_n\}_{n \in \mathbb{N}}$ *with* $X = \mathbb{N}$ *and* $a_0 := 0$ *and for all* $n \in \mathbb{N}_{\geqslant 1}$ *is* $a_n(\{\!\{x_1, \ldots, x_n\}\!\}) := (x_1 + \cdots + x_n)/n$*,*
- $\mathsf{max} = \{m_n\}_{n \in \mathbb{N}}$ *with* $X = \mathbb{N} \cup \{-\infty\}$ *and* $m_0 := -\infty$*, and for all* $n \in \mathbb{N}_{\geqslant 1}$ *is* $m_n(\{\!\{x_1, \ldots, x_n\}\!\}) := \max\{x_1, \ldots, x_n\}$*.*

We often write fx instead of $f(x)$ for all $x \in \{\!\{X\}\!\}$.

In the remainder of this section we give examples for queries with aggregates. A formal definition of the syntax and semantics of queries with aggregates is given in Section 6.3.

To describe queries with aggregates, we introduce expressions that describes how to aggregate over the database elements. Let us assume, we have the query and the database from Example 4.4. Now, we want to define a query, that for every $\overline{a} \in \varphi(D)$ we have the first three components of $\overline{a}$ and a fourth component which is the sum over every corresponding fourth component in $\varphi(D)$ for the first three components, i.e.

$$\left\{ (\alpha(y), \alpha(x_1), \alpha(x_2), n) \;:\; n = \sum_{\substack{\beta \supseteq \alpha|_{\{y, x_1, x_2\}}, \\ (D, \beta) \models \varphi}} \beta(x_3),\ (D, \alpha) \models \varphi \right\}. \tag{6.1}$$

y	x_1	x_2	$\mathsf{sum}(x_3)$
1	1	4	1
1	1	5	2
1	1	6	7
1	2	4	1
1	2	5	2
1	2	6	7
1	3	4	1
1	3	5	2
1	3	6	7
2	4	2	13
2	8	2	13
2	9	2	13
3	2	1	1

y	x_1	$\mathsf{max}(\mathsf{prod}(x_2, \mathsf{sum}(x_3)))$
1	1	42
1	2	42
1	3	42
2	4	26
2	8	26
2	9	26
3	2	1

y	$\mathsf{count}(x_1)$	$\mathsf{max}(\mathsf{prod}(x_2, \mathsf{sum}(x_3)))$
1	3	42
2	3	16
3	1	1

Table 6.1.: Enumeration of $Q_{\mathsf{sum}}(D_0)$, $Q_{\mathsf{max}}(D_0)$ and $Q_{\mathsf{count}}(D_0)$ on the database from Example 4.4.

To describe a query for the set (6.1), we can define

$$Q_{\mathsf{sum}} \;:=\; \{(y, x_1, x_2, \mathsf{sum}(x_3)) \;:\; Eyx_1 \;\wedge\; Fyx_2x_3 \;\wedge\; Gyx_2x_3\}$$

The query result of Q_{sum} on the database of Example 4.4 is given in Table 6.1. Another example is a query that generates triple such that for every $\overline{a} = (a_1, a_2, a_3, a_4) \in Q_{\mathsf{sum}}(D)$, we output the first component a_1 and the second component a_2 and as a third component the maximum of the product a_3 and a_4, i.e.,

$$\left\{ (\alpha(y), \alpha(x_1), n) \;:\; n = \max_{\substack{\gamma \supseteq \alpha|_{\{y,x_1\}},\\ (D,\gamma) \models \varphi}} \left\{ \gamma(x_2) \cdot \sum_{\substack{\beta \supseteq \gamma|_{\{y,x_1,x_2\}},\\ (D,\beta) \models \varphi}} \beta(x_3) \right\}, (D, \alpha) \models \varphi \right\}. \tag{6.2}$$

A query for the set (6.2) can be defined as:

$$Q_{\mathsf{max}} \;:=\; \{(y, x_1, \mathsf{max}(\mathsf{prod}(x_2, \mathsf{sum}(x_3)))) \;:\; Eyx_1 \;\wedge\; Fyx_2x_3 \;\wedge\; Gyx_2x_3\}.$$

Furthermore, we can create a query with count aggregates. For example, we can define a query Q_{count} that outputs, evaluated on a database D, all triples of the form (a_1, m, a_3) where the following condition holds.

- There is a a_2 such that $(a_1, a_2, a_3) \in Q_{\mathsf{max}}$ and
- m is the number of elements $a_2 \in \mathrm{adom}(D)$ with $(a_1, a_2, a_3) \in Q_{\mathsf{max}}$,

i.e., $Q_{\mathsf{count}}(D)$ is the set

$$\left\{(\alpha(x_1), m, \alpha(x_3)) \ : \ m = |\{\beta : \beta \supseteq \alpha|_{\{x_1,x_3\}}, (D,\beta) \models Q_{\mathsf{max}}\}|, \ (D,\alpha) \models Q_{\mathsf{max}}\right\}.$$

The query Q_{count} can be described as

$$Q_{\mathsf{count}} \ := \ \{(y, \mathsf{count}(x_1), \mathsf{max}(\mathsf{prod}(x_2, \mathsf{sum}(x_3)))) \ : \ Eyx_1 \ \wedge \ Fyx_2x_3 \ \wedge \ Gyx_2x_3\}.$$

See Table 6.1 for the query results of Q_{sum} and Q_{max} and Q_{count} on the database of Example 4.4. As we see in our examples all queries Q are of the form

$$\{(x_1, \cdots, x_\ell, \mathsf{expr}_1, \cdots, \mathsf{expr}_m, b_1, \ldots, b_\ell) \ : \ \varphi\}$$

The enumeration results are grouped by $x_1, \cdots, x_p$. The expressions $\mathsf{expr}_1, \cdots, \mathsf{expr}_m$ are expressions that contain aggregate functions, such as $\mathsf{sum}, \mathsf{max}, \mathsf{prod}, \mathsf{count}$ and variables. We will later give a formal definition of syntax and semantics of *aggregate expressions*. The expressions will be defined such that for all expressions expr_j there exists a variable $v_j \in \mathrm{vars}(Q)$ such that the expression is inductively defined over the height of the variables in the subtree of T_{free} induced on $\mathsf{succ}(v_j)$ where T_{free} is the subtree of the q-tree of Q induced on $\mathrm{free}(Q)$.

Note that the form of the queries depends on an existing q-tree of a query. A query that is not allowed is the following

$$Q_3 \ := \ \{(\mathsf{count}(y), x_1, x_2, x_3) \ : \ Eyx_1 \ \wedge \ Fyx_2x_3 \ \wedge \ Gyx_2x_3\}.$$

Such a query cannot be maintained under updates in our notion. Let us assume, there is a dynamic algorithm $\mathtt{A}$ that receives Q_3 and a σ-db D_0, computes within arbitrary preprocessing time a data structure that can be updated in time $t_u = \mathit{poly}(Q) \cdot O(n^{1-\varepsilon})$ and allows to enumerate $Q_3(D)$ with delay $t_d = O(n^{1-\varepsilon})$ for all $\varepsilon > 0$. Such an algorithm enumerates the set

$$\left\{(n, \alpha_1(x_1), \alpha(x_2), \alpha(x_3)) \ : \ n = \left|\left\{\beta \supseteq \alpha|_{\{x_1,x_2,x_3\}} : (D,\beta) \models \varphi\right\}\right|, \ (D,\alpha) \models \varphi\right\}$$

where $\varphi := Eyx_1 \wedge Fyx_2x_3 \wedge Gyx_2x_3$. If we delete the first component in every output tuple, we receive the set $\{(\alpha_1(x_1), \alpha(x_2), \alpha(x_3)) \ : \ (D,\alpha) \models \varphi\}$. This is the result set of the query $Q' = \{(x_1, x_2, x_3) \ : \ \exists y \varphi\}$. Note that we output these tuples without repetition. For a contradiction, let us assume that we output a tuple $\overline{a} = (a_1, a_2, a_3)$ twice. Then, $\mathtt{A}$ enumerates (n, a_1, a_2, a_3) and (m, a_1, a_2, a_3) with $n \neq m$.

Note that by definition of $Q_3(D)$, both tuples can not belong to the set. All in all, we have an algorithm that computes within arbitrary preprocessing time a data structure that can be updated in sublinear time that allows to enumerate the set $Q'(D)$ with sublinear delay. Since Q' is not q-hierarchical this is a contradiction to Theorem 3.13.

In the next section we give a formal model to evaluate aggregates under updates.

6.2. A model to evaluate aggregations

To evaluate aggregates under updates, we define the *aggregate implementation*. The model we introduce here relies on the concept of a MUD-algorithm described in [41].

An aggregate implementation is a triple $(\xi, \circ, \eta)$ where $\xi : \mathbf{dom} \to \mathcal{M}$ is a function that maps a database entry to a message, $\circ : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ is a binary operation on $\mathcal{M}$ such that $(\mathcal{M}, \circ)$ is an Abelian group and $\eta : \mathcal{M} \to \mathbf{dom}$ is a function that maps the message to a database entry. The main idea is that we use the aggregate implementation to evaluate aggregate functions in the following way. For a multiset $\{\!\{a_1, \ldots, a_n\}\!\}$ we use the function ξ to map the database entries to messages, then we connect them with the binary function $\circ$ and use the solution as input on η to receive an output. An aggregate implementation computes an aggregate function $\{f_n\}_{n\in\mathbb{N}}$ if for all $n \in \mathbb{N}_{\geqslant 1}$ and for all $a_1, \ldots, a_n \in \mathbf{dom}$ the following holds:

$$f_n(\{\!\{a_1, \ldots, a_n\}\!\}) \;=\; \eta(\xi(a_1) \circ \cdots \circ \xi(a_n))$$

and $f_0 = \eta(e)$ where e is the neutral element in the group $(\mathcal{M}, \circ)$.

Example 6.3. *We consider now examples for aggregate implementations where* $\mathbf{dom} = \mathbb{N}$*:*

- *Let* $\mathcal{I}_{\mathsf{sum}} := (\xi_{\mathsf{sum}}, \circ_{\mathsf{sum}}, \eta_{\mathsf{sum}})$ *with*
  - $\mathcal{M} = \mathbb{Z}$*,*
  - $\xi_{\mathsf{sum}}(a) := a$ *for all* $a \in \mathbf{dom}$*,*
  - $\circ_{\mathsf{sum}} = +$ *and*
  - $\eta_{\mathsf{sum}}(a) = a$*.*

  $\mathcal{I}_{\mathsf{sum}}$ *computes* sum*. Note that* $\eta_{\mathsf{sum}}(\xi_{\mathsf{sum}}(a_1) \circ_{\mathsf{sum}} \cdots \circ_{\mathsf{sum}} \xi_{\mathsf{sum}}(a_n)) = a_1 + \cdots + a_n \in \mathbb{N}$ *if* $a_1, \ldots, a_n \in \mathbb{N}$*.*
- *Let* $\mathcal{I}_{\mathsf{prod}} := (\xi_{\mathsf{prod}}, \circ_{\mathsf{prod}}, \eta_{\mathsf{prod}})$ *with*
  - $\mathcal{M} = \mathbb{Q}$*,*
  - $\xi_{\mathsf{prod}}(a) := a$ *for all* $a \in \mathbf{dom}$*,*
  - $\circ_{\mathsf{prod}}$ *is the multiplication operator of* $\mathbb{Q}$ *and*
  - $\eta_{\mathsf{prod}}(a) = a$*.*

  $\mathcal{I}_{\mathsf{prod}}$ *computes* prod*. Note that* $\eta_{\mathsf{prod}}(\xi_{\mathsf{prod}}(a_1) \circ_{\mathsf{prod}} \cdots \circ_{\mathsf{prod}} \xi_{\mathsf{prod}}(a_n)) = a_1 \cdot \;\cdots\; \cdot a_n \in \mathbb{N}$ *if* $a_1, \ldots, a_n \in \mathbb{N}$*.*
- *Let* $\mathcal{I}_{\mathsf{count}} := (\xi_{\mathsf{count}}, \circ_{\mathsf{count}}, \eta_{\mathsf{count}})$ *with*
  - $\mathcal{M} = \mathbb{Z}$*,*
  - $\xi_{\mathsf{count}}(a) := 1$ *for all* $a \in \mathbf{dom}$*,*
  - $\circ_{\mathsf{count}} = +$ *and*
  - $\eta_{\mathsf{count}}(a) = a$*.*

  $\mathcal{I}_{\mathsf{count}}$ *computes* count*.*
- *Let* $\mathcal{I}_{\mathsf{avg}} := (\xi_{\mathsf{avg}}, \circ_{\mathsf{avg}}, \eta_{\mathsf{avg}})$ *with*
  - $\mathcal{M} = \mathbb{Z}^2$*,*

- $\xi_{\mathsf{avg}}(a) := (a, 1)$ *for all* $a \in \mathbf{dom}$,
- $(a_1, n_1) \circ_{\mathsf{avg}} (a_2, n_2) = (a_1{+}a_2,\ n_1{+}n_2)$ *for all* $(a_1, n_1), (a_2, n_2) \in \mathcal{M}$ *and*
- $\eta_{\mathsf{avg}}((a, n)) = a/n$.

$\mathcal{I}_{\mathsf{avg}}$ *computes* avg.

- *For the aggregate function* max *it is not sufficient to simply "remember" the maximum element of a list. If we remove every occurrence of the maximum element in the list during update steps, there is no information what the current maximum element is. To get rid of that problem, we use the following idea. To define an aggregate implementation* $(\xi_{\mathsf{max}}, \circ_{\mathsf{max}}, \eta_{\mathsf{max}})$ *for the aggregate function* max *we set* $\mathcal{M}$ *as the set of functions that maps elements from* $\mathbb{N} \cup \{-\infty\}$ *to* $\mathbb{Z}$. *The main idea is that for a multiset* $\{\!\{a_1, \ldots, a_n\}\!\}$ *the current message is a function that maps every* $a \in \mathbb{N} \cup \{-\infty\}$ *to the number of occurrences of* a *in* $\{\!\{a_1, \ldots, a_n\}\!\}$. *We call such a function the occurrence function of* $\{\!\{a_1, \ldots, a_n\}\!\}$. *For every* $a \in \mathbb{N} \cup \{-\infty\}$, *the value* $\xi_{\mathsf{max}}(a)$ *is the function* f *with* $f(a) = 1$ *and* $f(b) = 0$ *for all* $b \in (\mathbb{N} \cup \{-\infty\}) \setminus \{a\}$. *This is the occurrence function of* $\{\!\{a\}\!\}$. *The binary operation* $\circ_{\mathsf{max}}$ *outputs on input of* $f, g \in \mathcal{M}$ *the function* u *such that* $u(a) = f(a) + g(a)$ *for all* $a \in \mathbb{N} \cup \{-\infty\}$. *In other words, if* f *is the occurrence function of* $\{\!\{a_1, \ldots, a_n\}\!\}$ *and* g *the occurrence function of* $\{\!\{b_1, \ldots, b_m\}\!\}$, *then the function* $f \circ_{\mathsf{max}} g$ *is the occurrence function of* $\{\!\{a_1, \ldots, a_n, b_1, \ldots, b_m\}\!\}$.

  It remains to show, that $(\mathcal{M}, \circ_{\mathsf{max}})$ *is an Abelian group.*

  - $\circ_{\mathsf{max}}$ *is associative. Let* $p, q, r \in \mathcal{M}$. *Then, for all* $a \in \mathbb{N} \cup \{-\infty\}$ *the following holds:* $[(p \circ_{\mathsf{max}} q) \circ_{\mathsf{max}} r](a) = [p \circ_{\mathsf{max}} q](a) + r(a) = (p(a) + q(a)) + r(a) = p(a) + (q(a) + r(a)) = p(a) + [q \circ_{\mathsf{max}} r](a) = [p \circ_{\mathsf{max}} (q \circ_{\mathsf{max}} r)](a)$ *and, in particular, is* $(p \circ_{\mathsf{max}} q) \circ_{\mathsf{max}} r = p \circ_{\mathsf{max}} (q \circ_{\mathsf{max}} r)$.
  - *The neutral element* $e \in \mathcal{M}$ *is the function* $e(a) = 0$ *for all* $a \in \mathbb{N} \cup \{-\infty\}$ *since for all* $q \in \mathcal{M}$ *and for all* $a \in \mathbb{N} \cup \{-\infty\}$ *holds* $[q \circ_{\mathsf{max}} e](a) = q(a) + e(a) = q(a)$.
  - *For every* $q \in \mathcal{M}$ *there is an inverse element* $q^{-1} \in \mathcal{M}$ *where* $q^{-1}(a) = -q(a)$ *for all* $a \in \mathbb{N} \cup \{-\infty\}$ *since* $[q \circ_{\mathsf{max}} q^{-1}](a) = q(a) + q^{-1}(a) = 0$ *for all* $a \in \mathbb{N} \cup \{-\infty\}$, *i.e.,* $q \circ_{\mathsf{max}} q^{-1} = e$.
  - $\circ_{\mathsf{max}}$ *is commutative. Let* $p, q \in \mathcal{M}$. *For all* $a \in \mathbb{N} \cup \{-\infty\}$ *let* $[p \circ_{\mathsf{max}} q](a) = p(a) + q(a) = q(a) + p(a) = [p \circ_{\mathsf{max}} q](a)$. *In particular,* $p \circ_{\mathsf{max}} q = q \circ_{\mathsf{max}} p$.

  The post-processing function returns the maximum $a \in \mathbb{N} \cup \{-\infty\}$ *that appears in the multiset, i.e.,*

$$\eta(q) \ := \ \begin{cases} \max\{a \,:\, a \in \mathbb{N} \cup \{-\infty\}, q(a) \neq 0\} & \text{if } q \neq e \\ -\infty & \text{otherwise} \end{cases}$$

  It is straightforward to see that $(\xi_{\mathsf{max}}, \circ_{\mathsf{max}}, \eta_{\mathsf{max}})$ *computes the aggregation function* max.

For an aggregate implementation $(\xi, \circ, \eta)$ a data structure that maintains the aggregation function under updates represents the current message $m_{\texttt{current}} \in \mathcal{M}$ and the value $\eta(m_{\texttt{current}})$, i.e., the value $\eta(m_{\texttt{current}}) = f_n(\{\!\!\{a_1, \ldots, a_n\}\!\!\})$ if $\{\!\!\{a_1, \ldots, a_n\}\!\!\}$ is the current multiset.

To initialise the data structure, we initialise $m_{\texttt{current}}$ to the neutral element of $(\mathcal{M}, \circ)$ and compute $\eta(m_{\texttt{current}})$. If we *insert* an element a to the multiset, we update $m_{\texttt{current}}$ to the result of $m_{\texttt{current}} \circ \xi(a)$ and compute the new value $\eta(m_{\texttt{current}})$. If we *delete* an element a from the multiset, we compute $\xi(a)^{-1}$ (the inverse element of $\xi(a)$ in the group $(\mathcal{M}, \circ)$) and update $m_{\texttt{current}}$ to the result of $m_{\texttt{current}} \circ \xi(a)^{-1}$ and compute the new value $\eta(m_{\texttt{current}})$. It is straightforward to verify that $\eta(m_{\texttt{current}})$ always represents the value $f_n(\{\!\!\{a_1, \cdots, a_n\}\!\!\})$.

To design suitable data structures for $\mathcal{I}_{\mathsf{sum}}$ and $\mathcal{I}_{\mathsf{prod}}$ and $\mathcal{I}_{\mathsf{count}}$ and $\mathcal{I}_{\mathsf{avg}}$ it is sufficient to simply declare variables that store the values $m_{\texttt{current}}$ and $\eta(m_{\texttt{current}})$.

For $(\xi_{\mathsf{max}}, \circ_{\mathsf{max}}, \eta_{\mathsf{max}})$ we represent $m_{\texttt{current}}$ via an ordered list $\mathcal{L}$ with the values $\{(a, m_{\texttt{current}}(a)) : a \in \mathbb{N} \cup \{-\infty\},\ m_{\texttt{current}}(a) \neq 0\}$. The values are ordered descending by the first value of the tuples, i.e., if (a_2, b_2) is the successor element of (a_1, b_1) in $\mathcal{L}$, then $a_1 > a_2$. To lookup, insert and remove elements in the list fast, we use AVL-trees (see [65]). This takes time $O(\log n)$ where n is the number of elements in the current multiset. To initialise the data structure, we simply initialise an empty list. This represents the neutral element of $(\mathcal{M}, \circ_{\mathsf{max}})$. To update the data structure for $m_{\texttt{current}}$ to $m_{\texttt{current}} \circ_{\mathsf{max}} p$ where $p \in \xi(\mathbb{N} \cup \{-\infty\}) \cup \xi^{-1}(\mathbb{N} \cup \{-\infty\})$ let $b \in \mathbb{N} \cup \{-\infty\}$ be the unique element with $p(b) \in \{1, -1\}$. Note that, since $p \in \xi(\mathbb{N} \cup \{-\infty\}) \cup \xi^{-1}(\mathbb{N} \cup \{-\infty\})$, it holds that $p(a) = 0$ for all $a \in (\mathbb{N} \cup \{-\infty\}) \setminus \{b\}$. Then, we do the following steps. We lookup b in the AVL-tree. If it is not present, we insert $(b, p(b))$ to $\mathcal{L}$. Otherwise, we modify (b, n) to $(b, n + p(b))$. If $n + p(b) = 0$ remove $(b, 0)$ from $\mathcal{L}$. The value of $\eta(m_{\texttt{current}})$ is the first component of the first element in the list. It is straightforward to verify, that after an operation the list $\mathcal{L}$ represents $m_{\texttt{current}}$ and $\eta(m_{\texttt{current}})$ is the maximum element $a \in \mathbb{N} \cup \{-\infty\}$ with $m_{\texttt{current}}(a) \neq 0$. In particular, it takes time $O(\log n)$ to compute $m_{\texttt{current}} \circ_{\mathsf{max}} p$ from $m_{\texttt{current}}$ and for an element $p \in \xi(\mathbb{N} \cup \{-\infty\}) \cup \xi^{-1}(\mathbb{N} \cup \{-\infty\})$.

For an aggregate implementation $(\xi, \circ, \eta)$ let t_ξ be the time it takes to compute $\xi(a)$ for an $a \in \mathbf{dom}$, $t_{\xi^{-1}}$ be the time it takes to compute $\xi^{-1}(a)$ for a $a \in \mathbf{dom}$, $t_\circ$ be the time to compute $m_{\texttt{current}} \circ p$ if we receive $m_{\texttt{current}}$ and $p \in \xi(\mathbf{dom}) \cup \xi^{-1}(\mathbf{dom})$ as input and t_η the time it takes to compute $\eta(m_{\texttt{current}})$. For an aggregation function $\{f_n\}_{n \in \mathbb{N}}$ let $t_a(\{f_n\}_{n \in \mathbb{N}})$ be defined as follows. Let $\mathcal{A}$ be the set of all aggregate implementations that compute $\{f_n\}_{n \in \mathbb{N}}$, then

$$t_a(\{f_n\}_{n \in \mathbb{N}}) \ := \min_{(\xi, \circ, \eta) \in \mathcal{A}} t_\xi + t_{\xi^{-1}} + t_\circ + t_\eta,$$

i.e., it is the time it takes to perform an update in the multiset for the aggregate.

For the aggregate functions in Example 6.2, it follows that $t_{\xi_{\mathsf{agg}}} = t_{\xi_{\mathsf{agg}}^{-1}} = O(1)$ for $\mathsf{agg} \in \{\mathsf{sum}, \mathsf{prod}, \mathsf{count}, \mathsf{avg}, \mathsf{max}\}$ and $t_{\eta_{\mathsf{agg}}} = t_{\circ_{\mathsf{agg}}} = O(1)$ for $\mathsf{agg} \in \{\mathsf{sum}, \mathsf{prod}, \mathsf{count}, \mathsf{avg}\}$ and $t_{\eta_{\mathsf{max}}} = t_{\circ_{\mathsf{max}}} \leqslant O(\log n)$. In particular is $t_a(\mathsf{max}) = O(\log n)$ and $t_a(\mathsf{agg}) = O(1)$ for $\mathsf{agg} \in \{\mathsf{sum}, \mathsf{prod}, \mathsf{count}, \mathsf{avg}\}$.

We will now give syntax and semantics of aggregation expressions.

6.3. Syntax and semantics of aggregation expressions

In this section, we give a precise definition of the syntax and semantic of aggregation expressions. The notion relies on the notion of conjunctive queries with aggregates [31] but we will define aggregate expressions that relies on the q-tree of the q-hierarchical query. Let AGGR be the set of aggregate functions.

Definition 6.4 (Syntax of aggregate expression)**.** *Let Q be a q-hierarchical query and let T be a q-tree of Q and T_{free} be the induced subtree of T on* $\text{free}(Q)$*. For all $v \in V$ we define the set $\mathsf{AGGEXPR}_v^{T_{\text{free}}}$ of aggregate expressions over v, inductively over the height of a node in T_{free}.*

- *Let v be a leaf in T_{free} and $\mathcal{F} \in \mathsf{AGGR}$. Then, $\mathcal{F}(v) \in \mathsf{AGGEXPR}_v^{T_{\text{free}}}$.*
- *Let v be a node of height $\geqslant 1$ in T_{free} and $\mathcal{F}, \mathcal{G} \in \mathsf{AGGR}$ and let $u_1, \ldots, u_s$ be the children of v in T_{free} and $\mathsf{expr}_j \in \mathsf{AGGEXPR}_{u_j}^{T_{\text{free}}}$ for all $j \in [s]$. Then,*
  - $\mathcal{G}(\mathcal{F}(\mathsf{expr}_1, \ldots, \mathsf{expr}_s)) \in \mathsf{AGGEXPR}_v^{T_{\text{free}}}$ *and*
  - $\mathcal{G}(\mathcal{F}(v, \mathsf{expr}_1, \ldots, \mathsf{expr}_s)) \in \mathsf{AGGEXPR}_v^{T_{\text{free}}}$ *and*
  - $\mathcal{G}(\langle v, \mathsf{expr}_1, \ldots, \mathsf{expr}_s \rangle) \in \mathsf{AGGEXPR}_v^{T_{\text{free}}}$

The last expression $\mathcal{G}(\langle v, \mathsf{expr}_1, \ldots, \mathsf{expr}_s \rangle)$ allows us to construct an aggregate that operates over the solution of all children of the current node (see Example 6.7).

If we consider the query φ from Example 4.4 and the corresponding q-tree T, and its induced subgraph T_{free} on $\text{free}(Q)$ then $\mathsf{sum}(x_3) \in \mathsf{AGGEXPR}_{x_3}^{T_{\text{free}}}$ and $\mathsf{count}(x_1) \in \mathsf{AGGEXPR}_{x_1}^{T_{\text{free}}}$ and $\mathsf{max}(\mathsf{prod}(x_2, \mathsf{sum}(x_3))) \in \mathsf{AGGEXPR}_{x_2}^{T_{\text{free}}}$.

Definition 6.5 (q-hierarchical query with aggregates)**.** *A q-hierarchical query with aggregates is a conjunctive query of the form*

$$Q \;:=\; \{(x_1, \ldots, x_s, \mathsf{expr}_1, \ldots, \mathsf{expr}_r, b_1, \ldots, b_\ell) \;:\; \exists x_{k+1} \cdots \exists x_m \varphi\}$$

such that the query

$$Q' = \{(x_1, \ldots, x_k, b_1, \ldots, b_\ell) \;:\; \exists x_{k+1} \cdots \exists x_m \varphi\}$$

with $s \leqslant k$ is q-hierarchical and $x_1, \ldots, x_k \in \text{vars}$ and $b_1, \ldots, b_\ell \in \mathbf{dom}$ and there is a q-tree T of Q' and an induced subtree T_{free} of T on $\text{free}(Q')$ *such that for all $j \in [r]$ is $\mathsf{expr}_j \in \mathsf{AGGEXPR}_{v_j}^{T_{\text{free}}}$ with $v_j \in \text{free}(\varphi)$ and*

$$\{x_1, \ldots, x_s\} \cup \bigcup_{j=1}^{r} \mathit{succ}(v_j) \;=\; \text{free}(Q')\,.$$

The q-tree of Q is the q-tree of Q' and $\text{free}(Q) := \text{free}(Q')$.

The queries Q_{sum}, Q_{max} and Q_{count}, we considered at the beginning of this chapter, are examples for q-hierarchical queries with aggregates.

Now, we give a precise definition of the semantics of an aggregate expression.

Definition 6.6 (Semantics of an aggregate expression)**.** *Let Q be a q-hierarchical query with aggregates, let T be a q-tree of Q and let T_{free} be the induced subgraph of T on* free(Q).

For all σ-databases D and all $v \in V(T_{\text{free}})$ and all $\alpha : \text{vpath}[v) \to \text{adom}(D)$ let $[\![\cdot]\!]^{(D,\alpha)} : \mathsf{AGGEXPR}_v^{T_{\text{free}}} \to \mathbf{dom}$ be a function that is defined inductively over the height of v as follows:

- *If v is a leaf in T_{free} and $\mathcal{F} = \{f_n\}_{n\in\mathbb{N}} \in \mathsf{AGGR}$. Then,*

$$[\![\mathcal{F}(v)]\!]^{(D,\alpha)} \;:=\; f_n\left\{\!\!\left\{ a^\iota : \iota \in \mathcal{L}_v^{[\alpha]} \right\}\!\!\right\}$$

  The variable of $\mathcal{F}(v)$ is v.

- *If v is a node of height $h \geqslant 1$ in T_{free} and let $u_1, \ldots, u_s$ be the children of v and $\mathcal{F} = \{f_n\}_{n\in\mathbb{N}} \in \mathsf{AGGR}$ and $\mathcal{G} = \{g_n\}_{n\in\mathbb{N}} \in \mathsf{AGGR}$. Then,*

$$\begin{aligned}&[\![\mathcal{G}(\mathcal{F}(\mathsf{expr}_1, \ldots, \mathsf{expr}_s))]\!]^{(D,\alpha)} \;:=\\ &\quad g_n\left\{\!\!\left\{ f_s\left\{\!\!\left\{ [\![\mathsf{expr}_j]\!]^{(D,\alpha^\iota)} \;:\; j \in [s] \right\}\!\!\right\} : \iota \in \mathcal{L}_v^{[\alpha]} \right\}\!\!\right\}\end{aligned}$$

$$\begin{aligned}&[\![\mathcal{G}(\mathcal{F}(v, \mathsf{expr}_1, \ldots, \mathsf{expr}_s))]\!]^{(D,\alpha)} \;:=\\ &\quad g_n\left\{\!\!\left\{ f_{s+1}\left(\{\!\!\{ a^\iota \}\!\!\} \cup \left\{\!\!\left\{ [\![\mathsf{expr}_j]\!]^{(D,\alpha^\iota)} \;:\; j \in [s] \right\}\!\!\right\} \right) : \iota \in \mathcal{L}_v^{[\alpha]} \right\}\!\!\right\}\end{aligned}$$

$$\begin{aligned}&[\![\mathcal{G}(\langle v, \mathsf{expr}_1, \ldots, \mathsf{expr}_s\rangle)]\!]^{(D,\alpha)} \;:=\\ &\quad g_n\left\{\!\!\left\{ \langle a^\iota, [\![\mathsf{expr}_1]\!]^{(D,\alpha^\iota)}, \ldots, [\![\mathsf{expr}_s]\!]^{(D,\alpha^\iota)}\rangle : \iota \in \mathcal{L}_v^{[\alpha]} \right\}\!\!\right\}\end{aligned}$$

  The variable of these expressions above is v.

For all databases D and all queries with aggregates

$$Q = \{(x_1, \ldots, x_m, \mathsf{expr}_1, \ldots, \mathsf{expr}_r, b_1, \ldots, b_\ell) \;:\; \varphi\}$$

is

$$Q(D) \;:=\; \left\{ \begin{array}{l} (\alpha(x_1), \ldots, \alpha(x_m), n_1, \ldots, n_r, b_1, \ldots, b_\ell) \;: \\ \qquad n_j = [\![\mathsf{expr}_j]\!]^{(D,\alpha|_{\text{vpath}[v_j)})} \text{ for all } j \in [r] \text{ and } (D,\alpha) \models \varphi \end{array} \right\}.$$

A *single result* query with aggregates is a query of the form

$$Q = \{(\mathsf{expr}) \;:\; \varphi\}$$

where expr is of the form $\mathcal{F}(\mathsf{expr}_1, \cdots, \mathsf{expr}_s)$ with $\mathcal{F} \in \mathsf{AGGR}$ and $\mathsf{expr}_j \in \mathsf{AGGEXPR}_{u_j}^{T_{\text{free}}}$ for all $j \in [s]$ and $u_1, \ldots, u_s$ are the children of v_{root} in T_{free}. For all σ-databases D let $Q(D) := f_s\left\{\!\!\left\{ [\![\mathsf{expr}_j]\!]^{(D,\emptyset)} \;:\; j \in [s] \right\}\!\!\right\}$.

The aggregation time t_a of a query with aggregates Q is defined as $t_a(Q) := \max_{\mathcal{F} \in \mathcal{A}} t_a(\mathcal{F})$ where $\mathcal{A}$ is the set of the aggregation function that occur in the expressions $\mathsf{expr}_1, \ldots, \mathsf{expr}_r$. For example, for the query Q_{count} is $t_a(Q_{\mathsf{count}}) = O(\log(n))$ and for the query Q_{sum} is $t_a(Q_{\mathsf{sum}}) = O(1)$.

Example 6.7. *Recall the scenario from Example 6.1. We will now give queries for the statements given in the example:*

- *Output name and id of the people and the sum of their salaries:*

$$Q_1 \;=\; \{(x_{pid}, x_{name}, \mathsf{sum}(x_{salary})) \;:\; \exists x_{project}\; \varphi\}$$

where

$$\varphi \;:=\; \mathit{Person}(x_{pid}, x_{name}) \;\wedge\; \mathit{Salary}(x_{pid}, x_{project}, x_{salary})$$

- *Output the average salary of each individual project.*

$$Q_2 \;=\; \{(x_{project}, \widetilde{\mathsf{avg}}(\langle x_{salary}, \mathsf{count}(x_{pid})\rangle)) \;:\; \mathit{Salary}(x_{pid}, x_{project}, x_{salary})\}$$

where $\widetilde{\mathsf{avg}} = (a_n)_{n \in \mathbb{N}}$ *is the aggregate*

$$a_n(\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle) \;:=\; \frac{\sum_{i \in [n]} x_i y_i}{\prod_{i \in [n]} y_i}.$$

- *Output the project with highest total salary:*

$$Q_3 \;=\; \{(\mathsf{max}_2(\langle x_{project}, \mathsf{sum}(x_{salary})\rangle)) \;:\; \exists x_{pid} \mathit{Salary}(x_{pid}, x_{project}, x_{salary})\}$$

where $\mathsf{max}_2 := \{m_n\}_{n \in \mathbb{N}}$ *is defined as*

$$m_n(\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle) = y_i$$

where i *is an index such that* $x_i = \max_{j \in [n]} x_j$.

An aggregate implementation $(\xi_{\widetilde{\mathsf{avg}}}, \circ_{\widetilde{\mathsf{avg}}}, \eta_{\widetilde{\mathsf{avg}}})$ for $\widetilde{\mathsf{avg}}$ is the following: Let $\mathcal{M} := \mathbb{Z} \times \mathbb{Q}$. For every $\langle x, y\rangle$ is $\xi_{\widetilde{\mathsf{avg}}}(\langle x, y\rangle) = (x \cdot y, y)$ and for all $(z_1, q_1), (z_2, q_2) \in \mathbb{Z} \times \mathbb{Q}$ is $(z_1, q_1) \circ_{\widetilde{\mathsf{avg}}} (z_2, q_2) = (z_1 + z_2, q_1 * q_2)$. The postprocessing function is $\eta_{\widetilde{\mathsf{avg}}}((z, q)) = z/q$. It is easy to verify that the aggregate implementation $(\xi_{\widetilde{\mathsf{avg}}}, \circ_{\widetilde{\mathsf{avg}}}, \eta_{\widetilde{\mathsf{avg}}})$ computes $\widetilde{\mathsf{avg}}$.

An aggregate implementation for max_2 can be constructed very similar to the aggregate implementation for max.

In Section 6.4 we show how to compute the q-tree of a q-hierarchical conjunctive query with aggregates is q-hierarchical. In the Section 6.5 we give an application for queries with aggregates. We show how to modify a q-hierarchical conjunctive query Q to a Boolean q-hierarchical query Q_c with aggregates such that $Q_c(D)$ is the number of tuples in the result set $Q(D)$, i.e., $Q_c(D) = |Q(D)|$. Afterwards, we show, in the next section, how to enrich the data structure described in Section 4.1 to maintain queries with aggregates. Finally, we show how to reduce queries with aggregates to queries without aggregates such that we obtain routines like **enumerate**, **test** and **count** for queries with aggregates.

6.4. How to compute the q-tree of a q-hierarchical conjunctive query with aggregates

In this section we show how to compute the q-tree of a q-hierarchical conjunctive query with aggregates.

Lemma 6.8. *Let*

$$Q \;=\; \{(x_1,\ldots,x_s,\mathsf{expr}_1,\ldots,\mathsf{expr}_r,b_1,\ldots,b_\ell) \;:\; \varphi\}$$

be a query where φ is a conjunctive formula, expr_j for all $j \in [r]$ is an aggregate expression, $x_1,\ldots,\overline{x}_s \in \text{vars}$ and $b_1,\ldots,b_\ell \in \mathbf{dom}$. There is an algorithm that tests in time poly(Q) whether Q is a q-hierarchical CQ with aggregates and, if it is the case, computes the q-tree of Q.

The remainder of this section is devoted to the proof of 6.8

Note that we defined the expressions over the q-tree of the aggreagte expression free version Q'. Thus, we have to find out if there is a q-tree for the query

$$Q' \;=\; \{(x_1,\ldots,x_s,\mathsf{expr}_1,\ldots,\mathsf{expr}_r,b_1,\ldots,b_\ell) \;:\; \varphi\}$$

that coincides with the expressions given in the query Q. In order to compute a corresponding q-tree we compute a tree T_{expr_q} for the aggregates expression expr_q for all $q \in [r]$. The tree T_{expr_q} is defined inductively.

- Let expr be an expression of the form $\mathcal{F}(v)$, then the tree T_{expr} is the tree that consists of the node v and has no edge, i.e., $V(T_{\mathsf{expr}}) = \{v\}$ and $E(T_{\mathsf{expr}}) = \emptyset$.
- Let expr be an expression of the form $\mathcal{G}(\mathcal{F}(\mathsf{expr}_1,\ldots,\mathsf{expr}_s))$, $\mathcal{G}(v,\mathcal{F}(\mathsf{expr}_1,\ldots,\mathsf{expr}_s))$ or $\mathcal{G}(\langle\mathsf{expr}_1,\ldots,\mathsf{expr}_s\rangle)$. Then, the tree T_{expr}, is the following.

$$V(T_{\mathsf{expr}}) \;:=\; V(T_{\mathsf{expr}_1}) \cup \cdots \cup V(T_{\mathsf{expr}_s}) \cup \{v\}$$
$$E(T_{\mathsf{expr}}) \;:=\; E(T_{\mathsf{expr}_1}) \cup \cdots \cup E(T_{\mathsf{expr}_s}) \cup \left\{(v,v_j) \;:\; \begin{array}{c} v_j \text{ is the variable of} \\ \mathsf{expr}_j \text{ for all } j \in [r] \end{array}\right\}$$

Note that it follows from the definition of aggregate expression that T_{expr_j} has to be a subtree in the q-tree of Q for all $j \in [r]$. In the following we define v_j as the variable of expr_j for all $j \in [r]$. If there are $j, j' \in [r]$ with $v_j \in \mathsf{succ}(v_{j'})$, then $T_{\mathsf{expr}_j} = T_{\mathsf{expr}_{j'}}|_{\mathsf{succ}(v_j)}$. Clearly, this can be tested in *poly*(Q). Let us now consider the set Φ_j of atoms in φ where the variable v_j appears. We will show in the following that for all $u \in \text{vars}(Q) \setminus V(T_{\mathsf{expr}_j})$ with $u \in \text{vars}(\Phi_j)$ we have that u occurs in every atom in Φ_j. Assume for a contradiction that there is a $u \in \text{vars}(Q) \setminus V(T_{\mathsf{expr}_j})$ with $\text{atoms}(v) \not\subseteq \text{atoms}(u)$. Because of the fact that Q is q-hierarchical we have that $\text{atoms}(u) \subset \text{atoms}(v)$ or $\text{atoms}(u) \subseteq \text{atoms}(v) = \emptyset$. Note that since $u \in \text{vars}(\Phi_j)$ and v appears in every atom in Φ_j we have that $\text{atoms}(u) \subseteq \text{atoms}(v) = \emptyset$ is not possible. If

atoms$(u) \subset$ atoms(v) it follows that $u \in \mathsf{succ}(v)$ and then $u \in V(T_{\mathsf{expr}_j})$ which violates $u \in \text{vars}(Q) \setminus V(T_{\mathsf{expr}_j})$.

Let $\widetilde{Q}$ be the query

$$\{(w_1, \ldots, w_p, b_1, \ldots, b_\ell) \ : \ \widetilde{\varphi}\}$$

we obtain if we modify Q in the following way.

- Remove every aggreagte expression.
- Remove in φ every variable that belongs to $\bigcup_{j=1}^{r} \mathsf{succ}(v_j)$. The conjunctive formula we obtain is φ'.
- $w_1, \ldots, w_p$ are the remaining free variables in $\widetilde{\varphi}$.

The algorithm for testing if Q is a valid q-hierarchical conjunctive query with aggregates works as follows.

1. For all $j \in [r]$ compute the tree T_{expr_r}.
2. For all $j, j' \in [r]$ test if $T_{\mathsf{expr}_j} = T_{\mathsf{expr}_{j'}}|_{\mathsf{succ}(v_j)}$ or $V(T_{\mathsf{expr}_j}) \cap T_{\mathsf{expr}_{j'}} = \emptyset$. If the result of one of these tests is false, return false.
3. Compute the query $\widetilde{Q}$ and test if $\widetilde{Q}$ is q-hierarchical. If the query is not q-hierarchical, return false.
4. Compute the q-tree $\widetilde{T}$ of $\widetilde{Q}$.
5. For all $j \in [r]$ such that there is no $j' \in [r] \setminus \{j\}$ with $v_{j'} \in \mathsf{succ}(v_j) \setminus \{v_j\}$ do the following. If $\text{vars}(\Phi_j) \setminus \mathsf{succ}(v_j) = \emptyset$, concatenate the tree T_{expr_j} to the root, i.e., add to $\widetilde{T}$ the nodes and the edges of T_{expr_j} and an edge from v_{root} to the root of T_{expr_j}. If $\text{vars}(\Phi_j) \setminus \mathsf{succ}(v_j) \neq \emptyset$, choose the variable of $u \in \text{vars}(\Phi_j) \setminus \mathsf{succ}(v_j)$ with the lowest hight in $\widetilde{T}$. Add the tree T_{expr_j} as an child of v, i.e., we add to $\widetilde{T}$ the nodes and edges of T_{expr_j} and an edge from u to the root of T_{expr_j}.
6. Test if $\widetilde{T}$ is a q-tree for Q'. Then, Q is a valid q-hierarchical conjunctive query with aggregates and $\widetilde{T}$ the the corresponding q-tree. Otherwise, the algorithm return false.

Note that every step in the algorithm can be done in $O(poly(Q))$. In the following claim we will show that the algorithm is correct.

Claim 6.9. *The algorithm returns* true *and a q-tree if and only if Q is a q-hierarchical conjunctive query with aggregates.*

Proof. The algorithm returns true if $\widetilde{T}$ (in the last step) is a q-tree for Q'. Note that T_{expr_j} is a subtree of $\widetilde{T}$. Thus, the query Q is q-hierarchical and $\widetilde{T}$ is the corresponding q-tree.

Let us now consider the case that the algorithm returns false. If the algorithm returns false in Step 2, there are $j, j' \in [r]$ with $T_{\mathsf{expr}_j} \neq T_{\mathsf{expr}_{j'}}|_{\mathsf{succ}(v_j)}$ and $V(T_{\mathsf{expr}_j}) \cap$

$V(T_{\mathsf{expr}_{j'}}) \neq \emptyset$. It follows form the definition of aggregate expressions that T_{expr_j} and $T_{\mathsf{expr}_{j'}}$ can not appear in the same q-tree. Therefore, the query is not q-hierarchical.

If the algorithm returns false in Step 3 the query $\widetilde{Q}$ is not q-hierarchical. Then, there are two variables $x, y \in \text{vars}(\widetilde{Q})$ with $\text{atoms}(x) \not\subseteq \text{atoms}(y)$, $\text{atoms}(y) \not\subseteq \text{atoms}(x)$ and $\text{atoms}(x) \cap \text{atoms}(y) \neq \emptyset$. Then, we have that there is an atom $\psi_x \in \text{atoms}(x) \setminus \text{atoms}(y)$ and an atom $\psi_y \in \text{atoms}(y) \setminus \text{atoms}(x)$ and atom $\psi_{xy} \in \text{atoms}(x) \cap \text{atoms}(y)$. Query $\widetilde{Q}$ is obtained from Q by removing variables in the atom. Note that the variable that has been removed, are removed in every atom. Hence, there are atoms in Q with $\psi_x \in \text{atoms}(x) \setminus \text{atoms}(y)$ and an atom $\psi_y \in \text{atoms}(y) \setminus \text{atoms}(x)$ and atom $\psi_{xy} \in \text{atoms}(x) \cap \text{atoms}(y)$. Thus, Q' is not q-hierarchical.

If the algorithm returns false in step 6, $\widetilde{T}$ is not a valid q-tree. Assume for a contradiction that Q is q-hierarchical. Consider that case that $\widetilde{T}$ is not a q-tree since there is an atom ψ in Q where $\text{vars}(\psi) \cup \{v_{root}\}$ does not form a path in $\widetilde{T}$. Since Q is q-hierarchical there is a q-tree $\widehat{T}$ for Q'. In particular, $\text{vars}(\psi) \cup \{v_{root}\}$ forms a path in $\widehat{T}$. If there is a variable $v \in \{v_1, \ldots, v_r\} \cap \text{vars}(\psi)$ then by definition of the aggregate expressions, the path has the following form. Every node that appears before v in the path does not belong to $V(T_{\mathsf{expr}})$ (where expr is the expression where v is the variable of) and thus there is a path from v_{root} to u where u is the previous variable of v in the path. The path from v_{root} to u does also appear in the q-tree of $\widetilde{Q}$. Then, it will be followed by v. Since T_{expr} must be appear identical in $\widetilde{T}$ and in $\widehat{T}$, there must also be a path from v to the rest of the nodes in the path. This is a contradiction.

Let us consider the case that $\widetilde{T}$ violates (2) in Definition 4.1. Then $\text{free}(Q) \neq \emptyset$ and there are at least two connected components of $\widetilde{T}_{\text{free}}$ where $\widetilde{T}_{\text{free}}$ is the induced subgraph of $\widetilde{T}$ of $\text{free}(Q)$. Without loss of generality, let T_1 and T_2 be two connected components and let T_1 be the connected component with $v_{root} \in V(T_1)$.

Let u be the root of the subtree T_2. Let w be the parent node of u in $\widetilde{T}$. If $w \in \mathsf{succ}(v_j)$ for a $j \in [r]$ we have that a contradiction to the fact that Q is q-hierarchical since T_{expr_j} must be contained in the q-tree of Q and hence, the subtree of the q-tree of Q induced on $\text{free}(Q)$ has two connected components. If $w \notin \mathsf{succ}(v_j)$ for a $j \in [r]$, i.e., $w \notin \bigcup_{j=1}^{r} \mathsf{succ}(v_j)$ let us consider two cases.

- Case 1: $u \notin \bigcup_{j=1}^{r} \mathsf{succ}(v_j)$. Since the q-tree of $\widetilde{Q}$ is a subgraph of $\widetilde{T}$ this is a contradiction to the fact that $\widetilde{(Q)}$ is q-hierarchical. Thus, false has to be output in step 3.

- Case 2: $u \in \{v_1, \ldots, v_r\}$. Since u is free and $\text{atoms}(v) \subseteq \text{atoms}(v)$ it follows that in every path from v_{root} to u there is the quantified variable v. Thus, there can no q-tree for Q exists and thus, Q is not q-hierarchical.

□

This concludes the proof of Lemma 6.8.

6.5. Counting the number of tuples in $Q(D)$

In this section we show how to use queries with aggregates to output the number of tuples in the result set $Q(D)$. To realise this, we define inductively over the height of the variables in the induced subtree T_{free} of a q-tree of Q on free(Q) the expression c_v.

Definition 6.10 (Expression for Counting)**.** *Let Q be a q-hierarchical query and let T_{free} be the induced subtree of a q-tree of Q on* free(Q)*. If v is a leaf in T_{free}, then $c_v := \mathsf{count}(v)$. Otherwise, let $u_1, \ldots, u_\ell$ be the children of v in T_{free}. Then, $c_v := \mathsf{sum}(\mathsf{prod}(c_{u_1}, \ldots, c_{u_\ell}))$.*

In the next lemma, we show that the expression c_v describes the number of tuples in the result.

Lemma 6.11. *Let*

$$Q \;=\; \{(x_1, \ldots, x_k, b_1, \cdots, b_\ell) \;:\; \varphi\}$$

be a q-hierarchical query and let T_{free} be the subtree of a q-tree of Q induced on free(Q)*. Let $x_1, \cdots, x_m$ be the children of v_{root} in T_{free}. The query*

$$Q_c := \{(\mathsf{prod}(c_{x_1}, \ldots, c_{x_m})) \;:\; \varphi\}$$

describes the number of the tuples in $\varphi(D)$, i.e., $Q_c(D) = \{|Q(D)|\}$.

To prove Lemma 6.11, the following claims (Claim 6.12 and Claim 6.13) will be convenient.

Claim 6.12. *Let Q be a q-hierarchical query and let T_{free} be the induced subtree of the q-tree of Q on* free(Q) *and let D be a σ-db. Let ι be an arbitrary fit item in the data structure for Q on D. Then the following equation holds.*

$$\left|\widetilde{\mathcal{E}}^{\iota}\right| \;=\; \prod_{j=1}^{\ell} \left| \bigcup_{\widetilde{\iota} \in \mathcal{L}^{\iota}_{w_j}} \widetilde{\mathcal{E}^{\iota}} \right| \tag{6.3}$$

where $w_1, \ldots, w_\ell$ are the children of v^ι in T_{free}.

Proof. From Lemma 4.12 we obtain:

$$\begin{aligned}
\left|\widetilde{\mathcal{E}}^{\iota}\right| \;&=\; \left| \left\{ \bigcup_{j=1}^{\ell} \alpha_j \cup \alpha^{\iota} : \begin{array}{l} \text{for all } w_j \in \mathit{child}(u) \cap \text{free}(Q) \text{ there is a } \widetilde{\iota} \in \mathcal{L}^{\iota}_{w_j} \\ \text{such that } \alpha_j \in \widetilde{\mathcal{E}^{\iota}} \end{array} \right\} \right| \\
&\overset{(+)}{=} \left| \left\{ (\alpha_1, \cdots, \alpha_\ell) : \begin{array}{l} \text{for all } w_j \in \mathit{child}(u) \cap \text{free}(Q) \text{ there is a } \widetilde{\iota} \in \mathcal{L}^{\iota}_{w_j} \\ \text{such that } \alpha_j \in \widetilde{\mathcal{E}^{\iota}} \end{array} \right\} \right| \\
&\overset{(++)}{=} \left| \left\{ \alpha_1 \in \widetilde{\mathcal{E}^{\iota}} : \widetilde{\iota} \in \mathcal{L}^{\iota}_{w_1} \right\} \times \cdots \times \left\{ \alpha_\ell \in \widetilde{\mathcal{E}^{\iota}} : \widetilde{\iota} \in \mathcal{L}^{\iota}_{w_\ell} \right\} \right| \\
&= \prod_{j=1}^{\ell} \left| \left\{ \alpha_j \in \widetilde{\mathcal{E}^{\iota}} : \widetilde{\iota} \in \mathcal{L}^{\iota}_{w_j} \right\} \right| \;=\; \prod_{j=1}^{\ell} \left| \bigcup_{\widetilde{\iota} \in \mathcal{L}^{\iota}_{w_j}} \widetilde{\mathcal{E}^{\iota}} \right| .
\end{aligned}$$

In equation $(+)$ we use the following fact. Every assignment $\alpha \in \widetilde{\mathcal{E}}^{\hat{\iota}}$ can be identified with a tuple $(\alpha_1, \cdots, \alpha_\ell)$ where for all $w_j \in \mathit{child}(u) \cap \mathrm{free}(Q)$ there is a $\widetilde{\iota} \in \mathcal{L}^{\hat{\iota}}_{w_j}$ and an $\alpha_j \in \widetilde{\mathcal{E}}^{\widetilde{\iota}}$ with $\alpha_j \subseteq \alpha$, and, in particular, the number of assignments in $\widetilde{\mathcal{E}}^{\hat{\iota}}$ is equal to the number of tuples $(\alpha_1, \cdots, \alpha_\ell)$ of the form described above. In $(++)$ we rewrite the set to a Cartesian product. □

Claim 6.13. *Let Q be a q-hierarchical query and let T_{free} be the induced subtree of the q-tree of Q on $\mathrm{free}(Q)$ and let D be a σ-db. For all items i in the data structure of Q on D and for all $u \in \mathit{child}(v^i) \cap \mathrm{free}(Q)$ holds*

$$[\![c_u]\!]^{(D,\alpha^i)} = \left| \bigcup_{\iota \in \mathcal{L}^i_u} \widetilde{\mathcal{E}}^{\iota} \right|.$$

Proof. We show this claim inductively over the height of an item in T_{free}.

For the induction base, let us consider an item i of height 1. Then, for all $u \in \mathit{child}(v^i)$ the following holds.

$$[\![c_u]\!]^{(D,\alpha^i)} \;=\; [\![\mathsf{count}(u)]\!]^{(D,\alpha^i)} \;=\; \left|\left\{\!\!\left\{ a^\iota \;:\; \iota \in \mathcal{L}^i_u \right\}\!\!\right\}\right| \overset{(\star)}{=} \left| \bigcup_{\iota \in \mathcal{L}^i_u} \{a^\iota\} \right| \overset{(\star\star)}{=} \left| \bigcup_{\iota \in \mathcal{L}^i_u} \widetilde{\mathcal{E}}^{\iota} \right|$$

The equation marked with $(\star)$ follows from the fact that every element in the multiset appears once. The equations marked with $(\star\star)$ follows from the fact that $\widetilde{\mathcal{E}}^{\iota} = \{\alpha^\iota\}$ for all $\iota \in \mathcal{L}^i_u$.

For the inductive step, let us consider an item i of height > 1. For all $u \in \mathit{child}(v^i)$ which are leafs in T_{free} the claim can be shown analogously to the induction base. Let us now consider a child $u \in \mathit{child}(v^i)$ with height $\geqslant 1$. Let $w_1, \cdots, w_s$ be the children of u in T_{free}. Then, it follows that

$$\begin{aligned}
[\![c_u]\!]^{(D,\alpha^i)} &= [\![\mathsf{sum}(\mathsf{prod}(c_{w_1}, \ldots, c_{w_s}))]\!]^{(D,\alpha^i)} \\
&= s_n \left\{ p_s \left\{ [\![c_{w_j}]\!]^{(D,\alpha^\iota)} \;:\; j \in [s] \right\} : \iota \in \mathcal{L}^i_v \right\} \\
&= \sum_{\iota \in \mathcal{L}^i_v} \prod_{j=1}^{s} [\![c_{w_j}]\!]^{(D,\alpha^\iota)} \overset{(IH)}{=} \sum_{\iota \in \mathcal{L}^i_v} \prod_{j=1}^{s} \left| \bigcup_{\widetilde{\iota} \in \mathcal{L}^\iota_{w_j}} \widetilde{\mathcal{E}}^{\widetilde{\iota}} \right| \\
&\overset{(\star)}{=} \sum_{\iota \in \mathcal{L}^i_v} \left| \widetilde{\mathcal{E}}^{\iota} \right| \overset{(\star\star)}{=} \left| \bigcup_{\iota \in \mathcal{L}^i_u} \widetilde{\mathcal{E}}^{\iota} \right|
\end{aligned}$$

Equation (IH) follows from the induction hypothesis. The equation $(\star\star)$ follows from the fact that $\widetilde{\mathcal{E}}^{\iota_1} \cap \widetilde{\mathcal{E}}^{\iota_2} = \emptyset$ for two $\iota_1, \iota_2 \in \mathcal{L}^i_u$ with $\iota_1 \neq \iota_2$, i.e., the union in the last line is a disjoint union. The equation $(\star)$ follows from Claim 6.12.

This concludes the proof of Claim 6.13. □

Proof of Lemma 6.11. Note that the number of tuples in $\varphi(D)$ is equal to $\left|\widetilde{\mathcal{E}}^{[\emptyset]}\right|$. The aim is to show that $Q_c(D) = \left|\widetilde{\mathcal{E}}^{[\emptyset]}\right|$.

It follows that

$$Q_c(D) \;=\; p_s\left\{\left[\!\left[c_{x_j}\right]\!\right]^{(D,\emptyset)} \;:\; j \in [s]\right\} \;=\; \prod_{j=1}^{m}\left[\!\left[c_{x_j}\right]\!\right]^{(D,\emptyset)}$$

$$\overset{(6.13)}{=} \prod_{j=1}^{m}\left|\bigcup_{\iota\in\mathcal{L}^i_{x_j}}\widetilde{\mathcal{E}}^{\iota}\right| \overset{(6.12)}{=} \left|\widetilde{\mathcal{E}}^{[\emptyset]}\right|$$

This concludes the proof of Lemma 6.11. □

The next section shows how to enrich the data structure for q-hierarchical queries to maintain queries with aggregates.

6.6. The data structure for queries with aggregates

First of all, we give an example how to construct a data structure that maintains the query Q_{count} on the database given in Example 4.4. Recall the query Q_{count}:

$$Q_{\mathsf{count}} \;:=\; \{(y, \mathsf{count}(x_1), \mathsf{max}(\mathsf{prod}(x_2, \mathsf{sum}(x_3)))) \;:\; Eyx_1 \;\wedge\; Fyx_2x_3 \;\wedge\; Gyx_2x_3\}.$$

On every present item i in the data structure with $v^i = x_2$, we store two values $w_{\text{list}}^{\mathsf{sum}(x_3)}(i)$ and $w_{\text{item}}^{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(i)$ where

$$w_{\text{list}}^{\mathsf{sum}(x_3)}(i) \;:=\; \sum_{\hat{\imath}\in\mathcal{L}^i_{x_3}} a^{\hat{\imath}} \qquad \text{and} \qquad w_{\text{item}}^{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(i) \;:=\; a^i \cdot w_{\text{list}}^{\mathsf{sum}(x_3)}(i),$$

i.e., $w_{\text{list}}^{\mathsf{sum}(x_3)}(i)$ is the sum of the constants of the items in the x_3-list of i and $w_{\text{item}}^{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(i)$ stores the product of the sum and its constant. Furthermore, we store for every item i with $v^i = y$ a value $w_{\text{list}}^{\mathsf{count}(x_1)}(i)$ and $w_{\text{list}}^{\mathsf{max}(\mathsf{prod}(x_2,\mathsf{sum}(x_3)))}(i)$, where $w_{\text{list}}^{\mathsf{count}(x_1)}(i)$ is the number of items in the x_1-list of i and $w_{\text{list}}^{\mathsf{max}(\mathsf{prod}(x_2,\mathsf{sum}(x_3)))}(i)$ is the maximum over the $w_{\text{item}}^{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(\hat{\imath})$-values for all items $\hat{\imath}$ in the x_2-list of i, i.e.

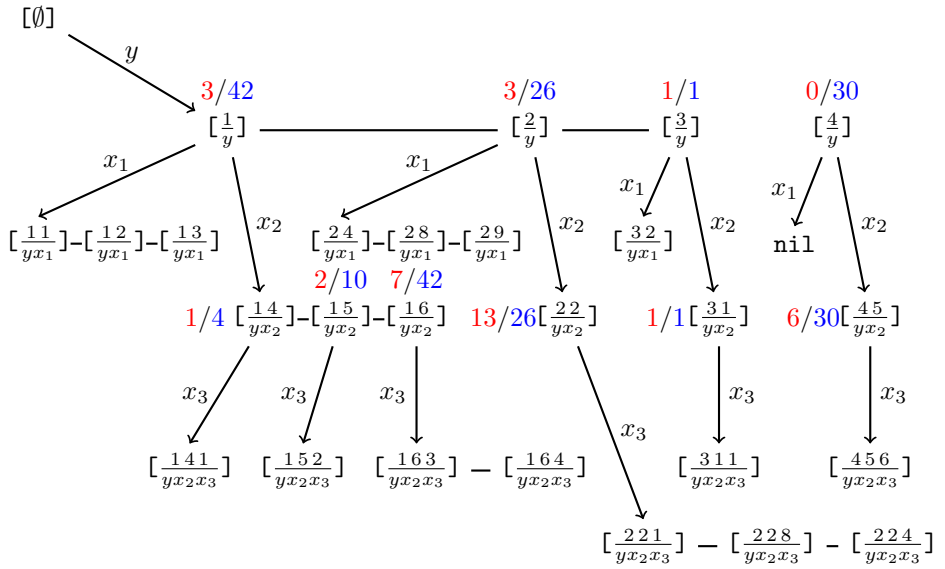
$$w_{\text{list}}^{\mathsf{count}(x_1)}(i) \;:=\; \left|\mathcal{L}^i_{x_1}\right|$$

and

$$w_{\text{list}}^{\mathsf{max}(\mathsf{prod}(x_2,\mathsf{sum}(x_3)))}(i) \;:=\; \max\left\{w_{\text{item}}^{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(\hat{\imath}) \;:\; \hat{\imath}\in\mathcal{L}^i_{x_2}\right\}.$$

See Figure 6.1 for an illustration of the data structure together with the values. Every red (blue) number on an item i with $v^i = x_2$ shows the value $w_{\text{list}}^{\mathsf{sum}(x_3)}(i)$ ($w_{\text{item}}^{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(i)$), the red (blue) number above an item i with $v^i = y$ shows the value $w_{\text{list}}^{\mathsf{count}(x_1)}(i)$ ($w_{\text{list}}^{\mathsf{max}(\mathsf{prod}(x_2,\mathsf{sum}(x_3)))}(i)$).

Figure 6.1.: Illustration of Example 4.4 with aggregates.



Now, suppose that $(4, 1)$ was inserted to E. As we know from our example the item $[\frac{4}{y}]$ is going to be fit and we insert $[\frac{41}{yx_1}]$ to the x_1-list of $[\frac{4}{y}]$. Therefore, we simply have to update the value $w_{\text{list}}^{\mathsf{count}(x_1)}(i)$ to 1. See Figure 6.2 for an illustration.

Different to the running example in Section 4.1 we now suppose that we delete $(1, 6, 4)$ from F. As a consequence, the item $[\frac{164}{yx_2x_3}]$ loses his fit-status. Therefore, we subtract the value 4 from $w_{\text{list}}^{\mathsf{sum}(x_3)}([\frac{16}{yx_2}])$ to obtain 3 and compute $w_{\text{item}}^{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}([\frac{1}{x_2}]) = 6 \cdot 3 = 18$. Now, we have to update $w_{\text{list}}^{\mathsf{max}(\mathsf{prod}(x_2,\mathsf{sum}(x_3)))}([\frac{1}{y}])$. We know that

$$w_{\text{list}}^{\mathsf{max}(\mathsf{prod}(x_2,\mathsf{sum}(x_3)))}([\frac{1}{y}]) = \max \{\!\{4, 10, 42\}\!\}$$

To obtain the new value, we compute from $\max \{\!\{4, 10, 42\}\!\}$ the value $\max \{\!\{4, 10, 42\}\!\} \setminus \{\!\{42\}\!\} = \max \{\!\{4, 10\}\!\}$ and then from $\max \{\!\{4, 10\}\!\}$ the value $\max \{\!\{4, 10, 18\}\!\}$. Note that this can be done in $O(\log n)$ since the aggregation time for max is at most $O(\log(n))$ where $n := |\text{adom}(D)|$.

See Figure 6.3 for an illustration after the update step.

Let us now go into technical details for arbitrary q-hierarchical queries with aggregates and databases. Let $Q = \{(x_1, \ldots, x_s, \mathsf{expr}_1, \ldots, \mathsf{expr}_r) \ : \ \varphi\}$ be the input query. Let T_Q be a q-tree of Q and let T_{free} be the subtree of T_Q induced on free(Q). Let D be a σ-db. For every $v \in V(T')$ let $\mathsf{AE}(v)$ be the set of subexpressions that appear in one of the expressions expr_t for $t \in [r]$ and expr_t is an expression with the variable v. To maintain queries with aggregates, we have to ensure that the following condition holds.

Figure 6.2.: Illustration of 4.4 with aggregates after $(4, 1)$ was inserted to E.

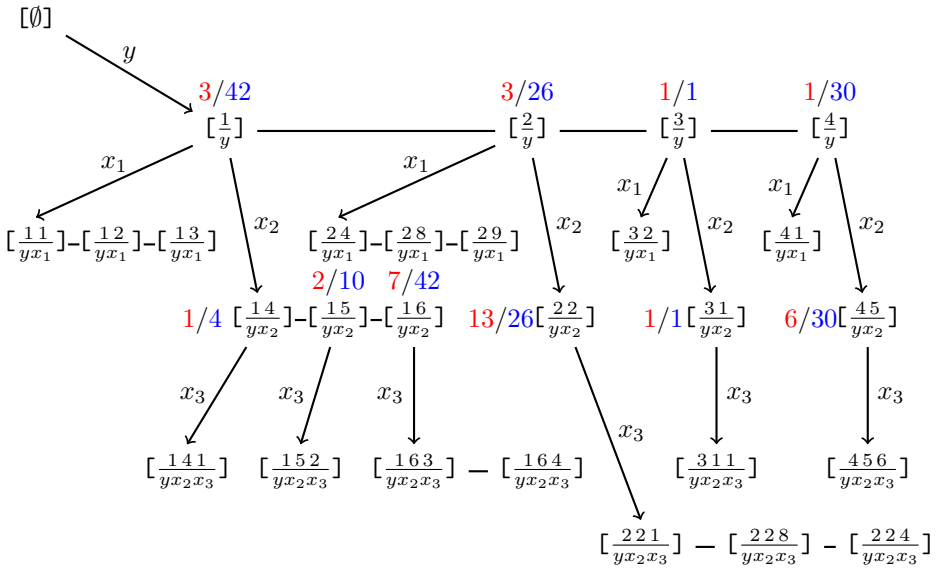
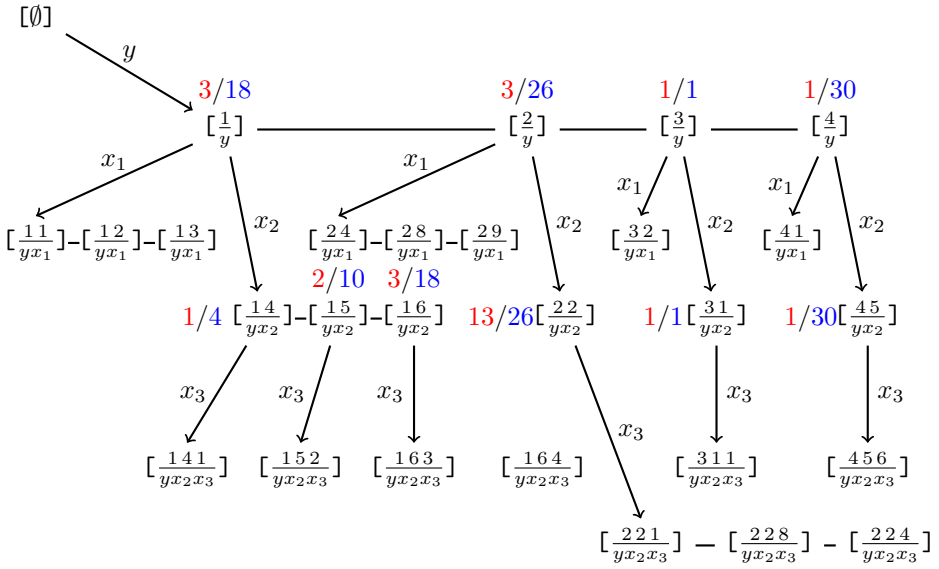


Figure 6.3.: Illustration of 4.4 with aggregates after $(4, 1)$ was inserted to E and $(1, 6, 4)$ deleted from F.



Condition 6.14. *For all present items i in the data structure for Q on D, where v^i is not a leaf in T_{free}, we store for every $u \in child(v^i)$ and every $\mathsf{expr} \in \mathsf{AE}(u)$ variables of the form $w_{list}^{\mathsf{expr}}(i)$ and $w_{item}^{\mathsf{expr}'}(i)$ (if expr is of the form $\mathcal{G}(\mathsf{expr}')$) such that the following is satisfied.*

- *if expr is of the form $\mathcal{F}(u)$ then*

$$w_{list}^{\mathcal{F}(u)}(i) \;=\; f_n\left\{\!\!\left\{ a^{\hat{\imath}} \;:\; \hat{\imath} \in \mathcal{L}_{u_j}^i \right\}\!\!\right\}$$

- *and if expr is of the form $\mathcal{G}(\mathsf{expr}')$ then*

$$w_{list}^{\mathcal{G}(\mathsf{expr}')}(i) \;=\; g_n\left\{\!\!\left\{ w_{item}^{\mathsf{expr}'}(\iota) : \iota \in \mathcal{L}_u^i \right\}\!\!\right\}$$

  whereas if expr' is of the form $\mathcal{F}(\mathsf{expr}_1, \ldots, \mathsf{expr}_{\ell'})$ then

$$w_{item}^{\mathsf{expr}'}(\iota) \;=\; f_{\ell'}\left\{\!\!\left\{ w_{list}^{\mathsf{expr}_j}(\iota) \;:\; j \in [\ell'] \right\}\!\!\right\},$$

  and if expr' is of the form $\mathcal{F}(u, \mathsf{expr}_1, \ldots, \mathsf{expr}_{\ell'})$ then

$$w_{item}^{\mathsf{expr}'}(\iota) \;=\; f_{\ell'+1}\left(\{\!\!\{ a^\iota \}\!\!\} \cup \left\{\!\!\left\{ w_{list}^{\mathsf{expr}_j}(\iota) \;:\; j \in [\ell'] \right\}\!\!\right\}\right),$$

  and if expr' is of the form $\langle u, \mathsf{expr}_1, \ldots, \mathsf{expr}_{\ell'}\rangle$ then

$$w_{item}^{\mathcal{G}(\mathsf{expr}')}(\iota) \;=\; \langle a^\iota, w_{list}^{\mathsf{expr}_1}(\iota), \ldots, w_{list}^{\mathsf{expr}_\ell}(\iota)\rangle$$

The main idea is that these values are partial solutions of the aggregate values in the sense that $w_{\text{list}}^{\mathsf{expr}}(i) = [\![\mathsf{expr}]\!]^{(D,\alpha^i)}$. These values can be computed using a bottom-up algorithm. Such a value for i has to be changed only if there is an item in one of the u-lists of i, which value was changed, or the fit status of i changes. Therefore, whenever the fit status of an item changes, we can change all the partial solutions of $w_{\text{list}}^{\mathsf{expr}}(i)$ and, afterwards we recompute the partial solution for the parent item. We recompute the partial solutions of the corresponding parent item from the item we changed the values the last time, again and again, until we receive an parent item that has no aggregate or the item is the *start*-item.

In the following lemma, we show that the values stored in $w_{\text{list}}^{\mathsf{expr}}(i)$ are the partial solutions.

Lemma 6.15. *Let Q be a q-hierarchical CQ with aggregates and D be a σ-db and T_{free} be the induced subgraph of the q-tree of Q on $\text{free}(Q)$. For every fit item i in the data structure for Q on D (that corresponds to T_Q), of height $\geqslant 1$ in T_{free} and every $u \in \mathsf{succ}(v^i)$ and every $\mathsf{expr} \in \mathsf{AE}(u)$ the following holds: $w_{list}^{\mathsf{expr}}(i) = [\![\mathsf{expr}]\!]^{(D,\alpha^i)}$.*

Proof. We prove this lemma by induction over the height of an item in T_{free}. For the induction base, let i be a fit item i of height 1 in T_{free}. Then, for every $u \in child(v^i)$ has the expression expr the form $\mathcal{F}(u)$. Then,

$$w_{\text{list}}^{\mathcal{F}(u)}(i) \;=\; f_n\left\{\!\!\left\{ a^{\hat{\imath}} \;:\; \hat{\imath} \in \mathcal{L}_u^i \right\}\!\!\right\} \;=\; [\![\mathsf{expr}]\!]^{(D,\alpha^i)} \;.$$

For the inductive step, let us consider an item i of height $h > 1$ and let $u \in \mathit{child}(v^i)$ arbitrary and expr be an arbitrary expression in $\mathsf{AE}(u)$. If expr is of the form $\mathcal{F}(u)$, the claim follows from the same argument as the induction base. If expr is of the form $\mathcal{G}(\mathcal{F}(\mathsf{expr}_1, \ldots, \mathsf{expr}_{s'}))$ where $s' := |\mathit{child}(u)|$ then

$$\begin{aligned} w_{\text{list}}^{\mathsf{expr}}(i) &= g_n\left\{\!\!\left\{ w_{\text{item}}^{\mathcal{F}(\mathsf{expr}_1,\ldots,\mathsf{expr}_{s'})}(\iota) : \iota \in \mathcal{L}_u^i \right\}\!\!\right\} \\ &= g_n\left\{\!\!\left\{ f_{s'}\left\{\!\!\left\{ w_{\text{list}}^{\mathsf{expr}_j}(\iota) \;:\; j \in [s'] \right\}\!\!\right\} : \iota \in \mathcal{L}_u^i \right\}\!\!\right\} \\ &\overset{(IH)}{=} g_n\left\{\!\!\left\{ f_{s'}\left\{\!\!\left\{ [\![\mathsf{expr}_j]\!]^{(D,\alpha^\iota)} \;:\; j \in [s'] \right\}\!\!\right\} : \iota \in \mathcal{L}_u^i \right\}\!\!\right\} \\ &= [\![\mathsf{expr}]\!]^{(D,\alpha^i)} \end{aligned}$$

Note that the equation marked with (IH) follows by the induction hypothesis. The case that expr is of the form $\mathcal{G}(\mathcal{F}(u, \mathsf{expr}_1, \ldots, \mathsf{expr}_{s'}))$ or $\mathcal{G}(\langle u, \mathsf{expr}_1, \ldots, \mathsf{expr}_{s'}\rangle)$ can be shown by an analogous way. □

To update the data structure we do the following. Every time we create a new item i during the update procedure, we set for all $u \in \mathsf{succ}(v^i)$ and for all $\mathsf{expr} \in \mathsf{AE}(u)$ the value f_0 if expr has the form $\mathcal{F}(u)$ and the value g_0 if expr has the form $\mathcal{G}(\mathsf{expr}')$. Whenever the the fit status of an item i is changed we execute Algorithm 9.

Algorithm 9 The fit status of $[\frac{v_1,\ldots,v_p}{b_1,\ldots,b_p}]$ has changed.

For all $j \in \{0, \ldots, p\}$ let $\iota_j = [\frac{x_1,\ldots,x_j}{b_1,\ldots,b_j}]$.
for all $\mathsf{expr} \in \mathsf{AE}(v_p)$ **do**
  if expr is of the form $\mathcal{F}(v_p)$ **then**
    $w_{\text{list}}^{\mathsf{expr}}(\iota_{p-1}) \leftarrow f_{n+1}\left\{\!\!\left\{ a^{\hat{\iota}} \;:\; \hat{\iota} \in \mathcal{L}_{v_p}^{\iota_{p-1}} \right\}\!\!\right\}$.
  if expr is of the form $\mathcal{G}(\mathsf{expr}')$ **then**
    $w_{\text{list}}^{\mathsf{expr}}(\iota_{p-1}) = g_{n+1}\left\{\!\!\left\{ w_{\text{item}}^{\mathsf{expr}'}(\hat{\iota}) : \hat{\iota} \in \mathcal{L}_{v_p}^{\iota_{p-1}} \right\}\!\!\right\}$.
for all $j = p-2$ **to** 0 **do** ▷ Here v_0 will be identified as v_{root}
  for all $\mathsf{expr} \in \mathsf{AE}(v_j)$ **do** ▷ expr is of the form $\mathcal{G}(\mathsf{expr}')$)
    Let $\mathsf{expr}_q \in \mathsf{AE}(v_{j+1})$ be the subexpression of expr.
    Let $\ell' := |\mathit{child}(v_{j+1})|$.
    if expr' is of the form $\mathcal{F}(\mathsf{expr}_1, \ldots, \mathsf{expr}_{\ell'})$ **then**
      Set $w_{\text{item}}^{\mathsf{expr}_q}(\iota_{j+1}) = f_{\ell'}\left\{\!\!\left\{ w_{\text{list}}^{\mathsf{expr}_s}(\iota_{j+1}) : s \in [\ell'] \right\}\!\!\right\}$.
    if expr' is of the form $\mathcal{F}(v_{j+1}, \mathsf{expr}_1, \ldots, \mathsf{expr}_{\ell'})$ **then**
      Set $w_{\text{item}}^{\mathsf{expr}_q}(\iota_{j+1}) = f_{\ell'}\left(\{\!\!\{ a^{\iota_{j+1}} \}\!\!\} \cup \left\{\!\!\left\{ w_{\text{list}}^{\mathsf{expr}_s}(\iota_{j+1}) : s \in [\ell'] \right\}\!\!\right\}\right)$.
    if expr' is of the form $\langle v_{j+1}, \mathsf{expr}_1, \ldots, \mathsf{expr}_{\ell'}\rangle$ **then**
      Set $w_{\text{item}}^{\mathsf{expr}_q}(\iota_{j+1}) = \langle a^{\iota_{j+1}}, w_{\text{list}}^{\mathsf{expr}_1}(\iota_{j+1}), \ldots, w_{\text{list}}^{\mathsf{expr}_{\ell'}}(\iota_{j+1})\rangle$.
    Set $w_{\text{list}}^{\mathsf{expr}}(\iota_j) = g_n\left(\left\{\!\!\left\{ w_{\text{item}}^{\mathsf{expr}_q}(\hat{\iota}) : \hat{\iota} \in \mathcal{L}_{v_{j+1}}^{\iota_j} \right\}\!\!\right\}\right)$

Lemma 6.16. *Let Q be a q-hierarchical CQ with aggregates and let D be a σ-db. For all present items i in the data structure for Q on D the following holds.*

1. *After the data structure is initialised for the empty database, Condition 6.14 holds for i and*
2. *if Condition 6.14 holds for i, the condition still holds after the data structure was updated and Algorithm 9 was applied.*

Proof. When initialising the data structure for the empty database, we initialise a start item with empty lists and nothing else. In particular, Condition 6.14 holds.

Let us suppose that we have a data structure for a database D such that for all items Condition 6.14 holds for all present items in the data structure. Now the fit status of an item changed (or it is created or deleted). Then, it is straightforward to verify that the algorithm ensures for all $j \in \{0, \dots, p\}$, all possible $w^{\mathsf{expr}}_{\text{list}}(\iota_j)$ and all $w^{\mathsf{expr}}_{\text{item}}(\iota_j)$ are correct where expr are appropriate aggregate expressions.

Let us assume for a contradiction that there is an item $\hat{\iota} \notin \{\iota_1, \dots, \iota_p\}$ where the $w^{\mathsf{expr}}_{\text{list}}(\hat{\iota})$ or the $w^{\mathsf{expr}}_{\text{item}}(\hat{\iota})$-value is not correct and for all $u \in \mathit{child}(v^{\hat{\iota}}) \cap \text{free}(Q)$ and all $\tilde{\iota} \in \mathcal{L}^{\hat{\iota}}_u$ is the $w^{\mathsf{expr}}_{\text{list}}(\tilde{\iota})$ and the $w^{\mathsf{expr}}_{\text{item}}(\tilde{\iota})$ value is correct where expr are appropriate aggregate expressions. Then the corresponding multisets for $w^{\mathsf{expr}}_{\text{list}}(\hat{\iota})$ and $w^{\mathsf{expr}}_{\text{item}}(\hat{\iota})$ must be changed. But this can only happen if an item gets fit or unfit. Since ι_p is the only item whose fit status was changed, this violates the fact that there is an item $\hat{\iota} \notin \{\iota_1, \dots, \iota_p\}$ where the $w^{\mathsf{expr}}_{\text{list}}(\hat{\iota})$ or the $w^{\mathsf{expr}}_{\text{item}}(\hat{\iota})$-value is not correct. In particular it follows that the values are correct for all present items. This concludes the proof. □

We now analyse the running time of Algorithm 9:

Lemma 6.17. *Let Q be a q-hierarchical CQ with aggregates and let t_a be the aggregation time of Q. Algorithm 9 takes time poly$(Q)t_a$ on a data structure for an arbitrary σ-db D.*

Proof. Let us first consider line 4. The item ι_p changes the fit status, therefore it is added to or removed from the v_p-list of ι_{p-1}. Therefore, the value stored in $w^{\mathsf{expr}}_{\text{list}}(\iota_{p-1})$ (before the fit status of ι_p changed) is equal to $f_{n-1}\left\{\!\!\left\{ a^{\hat{\iota}} \;:\; \hat{\iota} \in \mathcal{L}^i_{u_j} \setminus \{\iota_p\} \right\}\!\!\right\}$ or $f_{n+1}\left\{\!\!\left\{ a^{\hat{\iota}} \;:\; \hat{\iota} \in \mathcal{L}^i_{u_j} \cup \{\iota_p\} \right\}\!\!\right\}$. To obtain the correct $w^{\mathsf{expr}}_{\text{list}}(\iota_{p-1})$ value update the value in time $O(t_a(\mathcal{F}))$ by adding or removing a ι_p to/from the multiset. With the same trick, we can update the value in line 6. Let us now consider line 12, 14 and 16). Since the values for the arguments $a^{\iota_{j+1}}$ and $w^{\mathsf{expr}_s}_{\text{item}}(\widehat{\iota})$ for all $s \in [\ell']$ are already stored and the number of these arguments are bounded by $\ell' + 1$, it takes at most $(|\mathit{child}(v_{j+1})| + 1)t_a$ to compute $w^{\mathsf{expr}_q}_{\text{item}}(\iota_{j+1})$ from scratch. To update $w^{\mathsf{expr}}_{\text{list}}(\iota_j)$ in line 17 it takes at most $2t_a$ to update the value, since we have to remove the "old" $w^{\mathsf{expr}_q}_{\text{item}}(\iota_{j+1})$ and insert the "new" value. Since the number of aggregate expressions is bounded by $|Q|$ it follows for the running time of Algorithm 9:

$$|Q| \cdot 2t_a + \sum_{j=0}^{p-2} O(|Q|)\big[(|\mathit{child}(v_{j+1})| + 1)t_a + 2t_a\big] \leqslant \mathit{poly}(Q)t_a$$

□

All in all, we have shown that there is a data structure that maintains queries with aggregates under updates with update time $t_u = poly(Q)t_a$, since at most $|\text{vars}(Q)|$ items change the fit status during an update.

6.7. A reduction to q-hierarchical queries without aggregates

To obtain an enumeration algorithm for queries with aggregates, we define for every q-hierarchical CQ Q with aggregates a q-hierarchical CQ $\widetilde{Q}$ without aggregates and for every database D a database $\widetilde{D}$ such that $Q(D) = \widetilde{Q}(\widetilde{D})$. Then, we can use already known routines such as **enumerate**, **test** and **diff** for $\widetilde{Q}(\widetilde{D})$ to obtain corresponding results for $Q(D)$. Furthermore, we guarantee that whenever the database D receives an update we have to update the data structure for $\widetilde{Q}(\widetilde{D})$ very quickly (this will depend on the aggregation time of Q).

Before we go into technical details, we show on an example how the reduction works. Let us recall the example from Section 6.6. We consider the query

$$Q_{\mathsf{count}} := \{(y, \mathsf{count}(x_1), \mathsf{max}(\mathsf{prod}(x_2, \mathsf{sum}(x_3)))) : Eyx_1 \wedge Fyx_2x_3 \wedge Gyx_2x_3\}.$$

and the database from Example 4.4. See Figure 6.1 for an illustration of the data structure together with $\mathsf{count}(x_1)$ (the red values above the items i with $v^i = y$) and $\mathsf{prod}(x_2, \mathsf{sum}(x_3))$ (the blue values above the items i with $v^i = y$).

Now from Q_{count}, we construct the following query by replacing the aggregate expressions $\mathsf{sum}(x_3)$ and $\mathsf{prod}(x_2, \mathsf{sum}(x_3))$ by variables $x_{\mathsf{count}(x_1)}$ and $x_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}$. Furthermore we introduce for every aggregate expression relations $R_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}yx_{\mathsf{count}(x_1)}$ and $R_{\mathsf{count}(x_1)}yx_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}$. We obtain the following q-hierarchical query:

$$\widetilde{Q}_{\mathsf{count}} := \left\{ (y, x_{\mathsf{count}(x_1)}, x_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}) : \begin{array}{c} R_{\mathsf{count}(x_1)}yx_{\mathsf{count}(x_1)} \wedge \\ R_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}yx_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))} \end{array} \right\}$$

of schema $\widetilde{\sigma} = \{R_{\mathsf{count}(x_1)}, R_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}\}$. We will now define a $\widetilde{\sigma}$-db $\widetilde{D}$. For every $\mathsf{expr} \in \{\mathsf{count}(x_1), \mathsf{prod}(x_2, \mathsf{sum}(x_3))\}$ we define the binary relation R_{expr} as follows. There is a pair $(a, b) \in R_{\mathsf{expr}}(\widetilde{D})$ if and only if the item $i := [\frac{a}{y}]$ is present in the data structure for $Q(D)$ (here, we mean the data structure described in Section 6.6) and $\mathcal{L}_v^i \neq \emptyset$ where v is the variable of expr and $b = w_{\text{list}}^{\mathsf{expr}}(i)$. In our example, we obtain the following database.

$$\begin{aligned} R^{\widetilde{D}}_{\mathsf{count}(x_1)} &= \{(1,3),\ (2,3),\ (3,1)\} \\ R^{\widetilde{D}}_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))} &= \{(1,42),\ (2,26),\ (3,1),\ (4,30)\} \end{aligned}$$

See Figure 6.4 for an illustration of $\widetilde{Q}(\widetilde{D})$. A red dotted (blue dashed) arrow from an item i to an item ι depicts that ι is in the $x_{\mathsf{count}(x_1)}$-list ($x_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}$-list) of i.

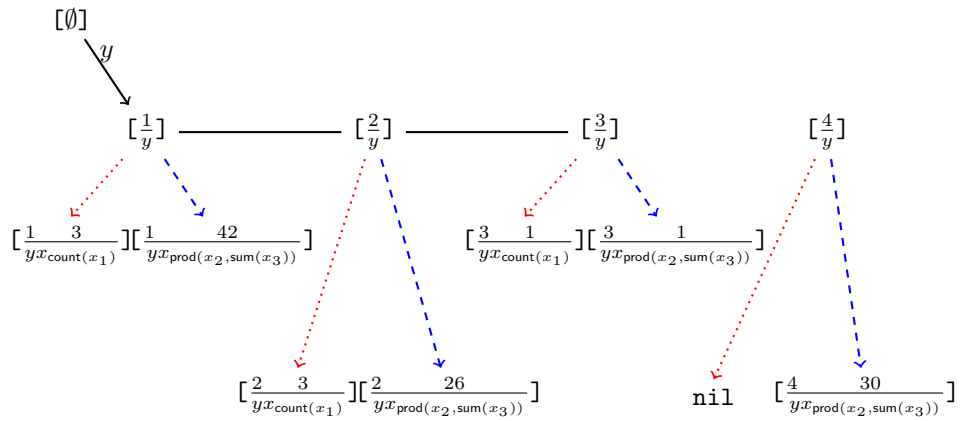


Figure 6.4.: Data structure for $\widetilde{Q}(\widetilde{D})$.

It is straightforward to see: When enumerating the tuples with the data structure in Figure 6.4, we get the result table shown in Table 6.1.

Let us assume that we insert $(4,1)$ to E. This forces that $\mathcal{L}_{x_1}^{[\frac{4}{y}]}$ is not empty after the update in the data structure for $\widetilde{Q}$ on $\widetilde{D}$. That is the reason why we have to insert the tuple $(4,1)$ to $R^{\widetilde{D}}_{\mathsf{count}(x_1)}$. We use the data structure from Section 6.6 to compute the new value $w^{\mathsf{count}(x_1)}_{\text{list}}([\frac{4}{y}])$ (see Figure 6.2 for the data structure after the update) and then we insert $(4,1)$ to $R^{\widetilde{D}}_{\mathsf{count}(x_1)}$ using the known algorithm for updating data structures for q-hierarchical queries. See Figure 6.5 for an illustration of the data structure.

As a last update step in our example, we delete $(1,6,4)$ from F. As a consequence, the value $w^{\mathsf{count}(x_1)}_{\text{list}}([\frac{1}{y}])$ changes from 42 to 18 (See Figure 6.3). Therefore, we delete the tuple $(1,42)$ from $R_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(\widetilde{D})$ and insert the tuple $(1,18)$ to $R_{\mathsf{prod}(x_2,\mathsf{sum}(x_3))}(\widetilde{D})$. See Figure 6.6 for the resulting data structure.

In the following lemma we will show that this idea works in general.

Lemma 6.18. *For every schema σ, every q-hierarchical query Q with aggregates of schema σ and every σ-db D, there is a schema $\widetilde{\sigma}$ and a q-hierarchical query $\widetilde{Q}$ without aggregates of schema $\widetilde{\sigma}$ and a $\widetilde{\sigma}$-db $\widetilde{D}$ such that the following conditions hold:*

(a) $Q(D) = \widetilde{Q}(\widetilde{D})$ and

(b) $\widetilde{Q}$ can be computed from Q in time $O(poly(Q))$ and

(c) an algorithm that maintains a data structure for $\widetilde{Q}$ on $\widetilde{\sigma}$-dbs with initialisation time $\widetilde{t_i}$ and update time $\widetilde{t_u}$ can be used to obtain an algorithm for maintaining a data structure for Q on σ-dbs with initialisation time $\widetilde{t_i}$ and update time $poly(Q)(\widetilde{t_u}+t_a)$ where t_a is the aggregation time of Q.

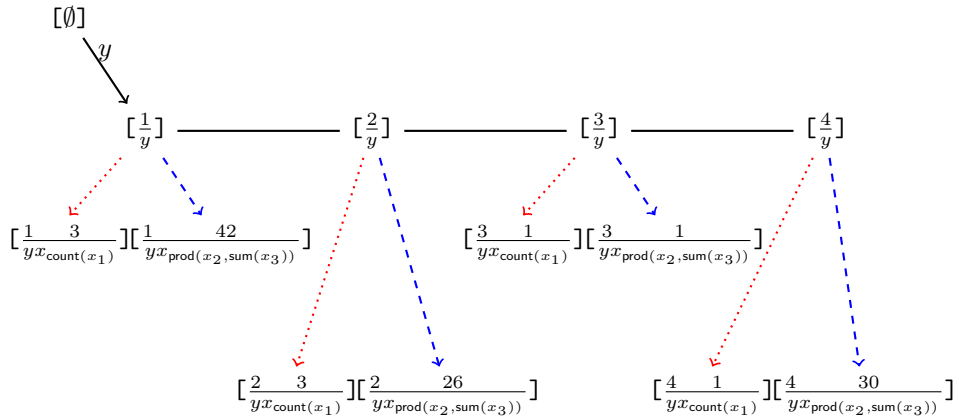


Figure 6.5.: Data structure for $\widetilde{Q}(\widetilde{D})$ after inserting $(4, 1)$ to E.

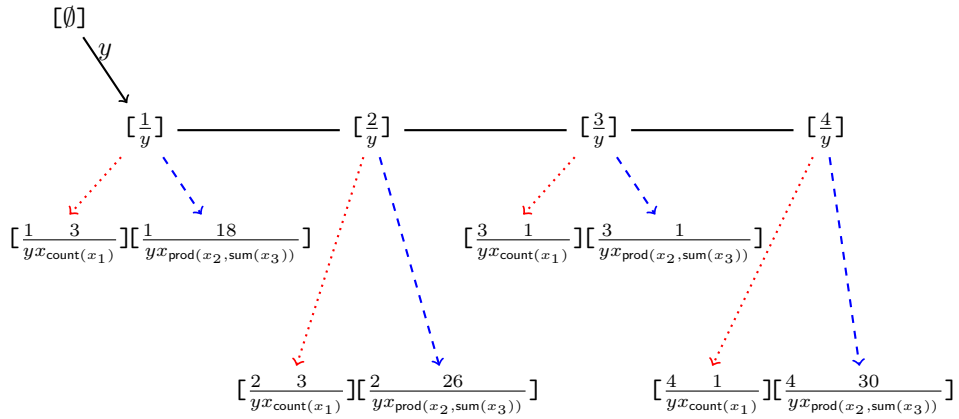


Figure 6.6.: Data structure for $\widetilde{Q}(\widetilde{D})$ after inserting $(4, 1)$ to E and deleting $(1, 6, 4)$ from F.

Proof. Let

$$Q \;=\; \{(x_1, \ldots, x_s, \mathsf{expr}_1, \ldots, \mathsf{expr}_r, b_1, \ldots, b_\ell) \;:\; \exists x_{k+1} \cdots \exists x_m \; \varphi\}$$

be the input query and let

$$Q' \;=\; \{(x_1, \ldots, x_k, b_1, \ldots, b_\ell) \;:\; \exists x_{k+1} \cdots \exists x_m \; \varphi\}$$

be the corresponding q-hierarchical query.

Definition of $\widetilde{\sigma}$, $\widetilde{Q}$ and $\widetilde{D}$. The schema $\widetilde{\sigma}$ consists of relations R_p for every $p \in [r]$ of arity $\mathrm{ar}(R_p) = |\mathrm{vpath}[v_p]|$ where v_p is the variable of expr_p and of relations $R \in \sigma$ such that there is an atom $Ry_1 \ldots y_{\mathrm{ar}(R)}$ in Q such that $\{y_1, \ldots, y_{\mathrm{ar}(R)}\} \cap \mathbf{var} \subseteq \{x_1, \ldots, x_s, x_{k+1}, \ldots, x_m\}$. Let $\widetilde{\varphi}$ be the conjunction with the following atoms

- every atom $\psi \in \mathrm{atoms}(Q)$ where $\mathrm{vars}(\psi) \subseteq \{x_1, \ldots, x_s, x_{k+1}, \ldots, x_m\}$ is an atom in $\widetilde{\varphi}$ and
- for every $p \in [r]$ there is an atom $R_p(u_1, \ldots, u_q, x_{\mathsf{expr}_p})$ in $\widetilde{Q}$ where $\{u_1, \ldots, u_q\} = \mathrm{vpath}[v_p)$ in which v_p is the variable of expr_p.

Let $x_{j_1}, \ldots, x_{j_q}$ be the variables in $\{x_{k+1}, \ldots, x_m\} \cap \mathrm{vars}(\widetilde{\varphi})$ and let $c_1, \ldots, c_p$ be the constants that appear in $\mathrm{atoms}(\varphi) \cap \mathrm{atoms}(\widetilde{\varphi})$. The query $\widetilde{Q}$ is the following:

$$\widetilde{Q} \;:=\; \{(x_1, \ldots, x_s, x_{\mathsf{expr}_1}, \ldots, x_{\mathsf{expr}_r}, c_1, \cdots, c_p) \;:\; \exists x_{j_1} \cdots \exists x_{j_q} \; \widetilde{\varphi}\}.$$

We now define the database $\widetilde{D}$. For all $R \in \sigma \cap \widetilde{\sigma}$ let $R(\widetilde{D}) := R(D)$ and for all $p \in [r]$ is

$$R_p(\widetilde{D}) \;:=\; \left\{ \begin{array}{l} (\alpha(u_1), \cdots, \alpha(u_q), w^{\mathsf{expr}}_{\mathrm{list}}(\mathtt{[}\alpha\mathtt{]})) : \\ \qquad \mathrm{dom}(\alpha) = \{u_1, \ldots, u_q\}, \text{ and} \\ \qquad \mathtt{[}\alpha\mathtt{]} \text{ is present in the data structure for } Q'(D), \text{ and} \\ \qquad \mathcal{L}^{\mathtt{[}\alpha\mathtt{]}}_{v_p} \neq \emptyset \text{ where } v_p \text{ is the variable of } \mathsf{expr}_p \end{array} \right\}.$$

The query $\widetilde{Q}$ is q-hierarchical. The q-tree for $\widetilde{Q}$ can be constructed as follows. Let T be a q-tree for Q' and let $T' = (V', E')$ be the subgraph of T induced on the set $\{v_{root}, x_1, \cdots, x_s, x_{j_1}, \ldots, x_{j_q}\}$. Note that by definition of queries with aggregates it follows that T' is connected. The tree $\widetilde{T} = (\widetilde{V}, \widetilde{E})$ is defined as follows

$$\widetilde{V} \;:=\; V' \,\dot\cup\, \{x_{\mathsf{expr}_p} : p \in [r]\}$$

$$\widetilde{E} \;:=\; E' \;\cup\; \{(w, x_{\mathsf{expr}_p}) : w \text{ is the parent node of the variable of } x_{\mathsf{expr}_p} \text{ in } T, p \in [r]\}$$

Clearly $\widetilde{V} = \mathrm{vars}(\widetilde{Q}) \cup \{v_{root}\}$. It remains to show that for every atom $\psi \in \mathrm{atoms}(\widetilde{Q})$ the set $\{v_{root}\} \cup \mathrm{vars}(\psi)$ forms a path in $\widetilde{T}$. If $\psi \in \mathrm{atoms}(Q)$, then $\mathrm{vars}(\psi) \subseteq \{x_1, \ldots, x_s, x_{j_1}, \ldots, x_{j_q}\}$ and $\{v_{root}\} \cup \mathrm{vars}(\psi)$ forms a path in T. Since T' is the induced subgraph of T on $x_1, \ldots, x_s, x_{j_1}, \ldots, x_{j_q}$ and T' is a subgraph of $\widetilde{T}$, it follows

that $\{v_{root}\} \cup \text{vars}(\psi)$ forms a path in $\widetilde{T}$. If $\psi \notin \text{atoms}(\widetilde{Q})$, there is a $p \in [r]$ such that ψ is of the form $R_p(u_1, \ldots, u_q, x_{\mathsf{expr}_p})$ where $\{u_1, \ldots, u_q\} = \text{vpath}[v_p)$ in which v_p is the variable of expr_p. Since $\{u_1, \ldots, u_q\} = \text{vpath}[v_p)$ the set $\{v_{root}, u_1, \ldots, u_q\}$ forms a path in T' and since $(u_q, x_{\mathsf{expr}_p}) \in \widetilde{E}$, it follows that $\{v_{root}\} \cup \text{vars}(\psi)$ forms a path in $\widetilde{T}$. Clearly, each of these paths starts from v_{root}. Since T' is a induced subgraph on $x_1, \ldots, x_s, x_{j_1}, \ldots, x_{j_q}$ and it is a subgraph of T, the set $x_1, \ldots, x_s$ is connected. Since for every $p \in [r]$ there is an edge (y, x_{expr_p}) for a $y \in \{x_1, \ldots, x_s\}$, the set $x_1, \ldots, x_s, x_{\mathsf{expr}_1}, \ldots, x_{\mathsf{expr}_q}$ is connected in $\widetilde{T}$. Therefore, the query $\widetilde{Q}$ is q-hierarchical.

Proof of part (a). As an abbreviation, we set α be an assignment such that $\overline{a} = (\alpha(x_1), \ldots, \alpha(x_s))$ and $\overline{c} := (c_1, \ldots, c_p)$.

$$
\begin{aligned}
\overline{s} \in Q(D) \;\Leftrightarrow\; & \overline{s} = (\overline{a}, n_1, \cdots, n_r, \overline{b}) \text{ where } n_j = [\![\mathsf{expr}_j]\!]^{(D, \alpha|_{\text{vpath}[v_j)})} \\
& \text{for all } j \in [r], \text{ and there is a } \beta \supseteq \alpha \\
& \text{with } (D, \beta) \models \varphi \text{ and } \text{dom}(\beta) = \text{vars}(Q') \\
\Leftrightarrow\; & \overline{s} = (\overline{a}, w_{\text{list}}^{\mathsf{expr}_1}(\texttt{[}\alpha|_{\text{vpath}[v_1)}\texttt{]}), \cdots, w_{\text{list}}^{\mathsf{expr}_r}(\texttt{[}\alpha|_{\text{vpath}[v_r)}\texttt{]}), \overline{b}) \\
& \text{and there is a } \beta \supseteq \alpha \text{ with} \\
& (D, \beta) \models \psi \text{ for all } \psi \in \text{atoms}(Q') \text{ and } \text{dom}(\beta) = \text{vars}(Q') \\
\overset{(\star)}{\Leftrightarrow}\; & \overline{s} = (\overline{a}, w_{\text{list}}^{\mathsf{expr}_1}(\texttt{[}\alpha|_{\text{vpath}[v_1)}\texttt{]}), \cdots, w_{\text{list}}^{\mathsf{expr}_r}(\texttt{[}\alpha|_{\text{vpath}[v_r)}\texttt{]}), \overline{c}) \\
& \text{and } (D, \alpha) \models \psi \text{ for all } \psi \in \text{atoms}(Q) \cap \text{atoms}(\widetilde{Q}) \\
& \text{and for all } R_p(u_1, \ldots, u_t, x_{\mathsf{expr}_p}) \in \text{atoms}(\widetilde{Q}) \\
& \text{holds } (\alpha(u_1), \ldots, \alpha(u_t), w_{\text{list}}^{\mathsf{expr}_p}(\texttt{[}\alpha|_{\text{vpath}[v_r)}\texttt{]})) \in R_p(\widetilde{D}) \\
\Leftrightarrow\; & \overline{s} \in \widetilde{Q}(\widetilde{D})
\end{aligned}
$$

To prove that equation $(\star)$ is correct, we show that there is a $\beta \supseteq \alpha$ such that $(D, \beta) \models \psi$ for all $\psi \in \text{atoms}(Q')$ if and only if $(D, \alpha) \models \psi$ for all $\psi \in \text{atoms}(Q') \cap \text{atoms}(\widetilde{Q})$ and for all atoms of the form $R_p(u_1, \ldots, u_r, x_{\mathsf{expr}_p})$ we have $(\alpha(u_1), \ldots, \alpha(u_r), w_{\text{list}}^{\mathsf{expr}_p}(\texttt{[}\alpha|_{\text{vpath}[v_r)}\texttt{]})) \in R_p(\widetilde{D})$.

For the "if" direction, note that $(D, \alpha) \models \psi$ for all $\psi \in \text{atoms}(Q') \cap \text{atoms}(\widetilde{Q})$ holds since $\text{vars}(\psi) \subseteq \text{dom}(\alpha)$ for all $\psi \in \text{atoms}(Q') \cap \text{atoms}(\widetilde{Q})$. It remains to show the second part. Let $R_p(u_1, \ldots, u_t, x_{\mathsf{expr}_p})$ be an arbitrary atom in $\text{atoms}(\widetilde{Q})$. Since there is a $\beta \supseteq \alpha$ such that $(D, \beta) \models \psi$ for all $\psi \in \text{atoms}(Q')$ it follows that $\texttt{[}\gamma\texttt{]}$ for all $\gamma \subseteq \alpha$, where $\text{dom}(\gamma)$ forms a path in T', is fit. It follows from Lemma 4.7 that for all $u \in \mathit{child}(v^{\texttt{[}\gamma\texttt{]}})$ that $\mathcal{L}_u^{\texttt{[}\gamma\texttt{]}}$ is not empty. Since the variable of expr_p is included in one of the sets $\mathit{child}(v^{\texttt{[}\gamma\texttt{]}})$ for one of those γ, it follows by definition of $\widetilde{D}$ that $(\alpha(u_1), \ldots, \alpha(u_r), w_{\text{list}}^{\mathsf{expr}_p}(\texttt{[}\alpha|_{\text{vpath}[v_r)}\texttt{]})) \in R_p(\widetilde{D})$.

We consider now the "only if" direction. Let $\widetilde{T}$ be the induced subgraph of $\widetilde{T}$ on $\{x_1, \ldots, x_s\} = \text{dom}(\alpha)$. We show by an induction over the height of the variables in $\widetilde{T}$ that the items $\iota_v := \texttt{[}\alpha|_{\text{vpath}[v]}\texttt{]}$ for all $v \in V(T') \cap V(\widetilde{T})$ are fit in the data structure

for Q' and D. Then, it follows that there is a $\beta \supseteq \alpha$ such that $(D, \beta) \models \psi$ for all $\psi \in \text{atoms}(Q')$.

For the induction base let us consider an item ι_v of height 0. Then, v is either a leaf in $\widetilde{T}$ or the successor nodes of v are the nodes from $\{x_1, \ldots, x_q\}$. Since $v \in \text{vars}(\widetilde{Q}) \cap \text{vars}(Q')$ we have $\text{exatoms}(v) \subseteq \text{atoms}(Q') \cap \text{atoms}(\widetilde{Q})$. Since $\text{exatoms}(v) \subseteq \text{atoms}(Q') \cap \text{atoms}(\widetilde{Q})$, it follows by assumption that for all $\psi \in \text{exatoms}(v)$ holds $(D, \alpha) \models \psi$. For every child $x_p \in \mathit{child}(v)$ ($\mathit{child}(v)$ might be empty) it follows that $x_p \in \{x_1, \ldots, x_q\}$ and there is a tuple in R_p with $(\alpha(u_1), \ldots, \alpha(u_r), n_p) \in R_p^{\widetilde{D}}$ where $v_{root}, u_1, \ldots, u_r$ is the path from v_{root} to x_p in T'. By definition, we have that $\mathcal{L}_{x_p}^{[\alpha|_{\text{vpath}[u]}]} \neq \emptyset$. It follows from Lemma 4.7 that ι_v is fit.

For the inductive step let us consider an item ι_v of height h. By the induction hypothesis we obtain that for all $u \in \mathit{child}(v)$ the item ι_u is fit and in particular, the lists $\mathcal{L}_u^{\iota_v}$ are not empty. Since $\text{exatoms}(v) \subseteq \text{atoms}(Q') \cap \text{atoms}(\widetilde{Q})$, it follows by assumption that for all $\psi \in \text{exatoms}(v)$ holds $(D, \alpha) \models \psi$. It follows from Lemma 4.7 that ι_v is fit.

Proof of part (b). It is straightforward to see that such a query can be easily computed in time $O(\mathit{poly}(Q))$ after the q-tree of Q was computed.

Proof of part (c). To maintain Q on D under updates, we use two data structures in parallel. The data structure $\mathtt{D}_1$ is the data structure in Section 6.6 for Q on D and a data structure $\mathtt{D}_2$ for evaluation $\widetilde{Q}$ on $\widetilde{\sigma}$-db $\widetilde{D}$ (with initialisation time t_i and update time t_u).

When *initialising* the data structure for the empty database, we initialise $\mathtt{D}_1$ and $\mathtt{D}_2$ for the empty database. This takes time $O(t_i)$.

When the database D receives an update, we update the data structure $\mathtt{D}_1$. If the update command contains a relation $R \in \sigma \cap \widetilde{\sigma}$, we update $\mathtt{D}_2$. During the update in $\mathtt{D}_1$, we store for every item i where a $w_{\text{list}}^{\mathsf{expr}_p}(i)$-value was changed, the old value $\mathsf{old}(w_{\text{list}}^{\mathsf{expr}_p}(i))$ (or $\mathtt{nil}$ if it does not exists) and the new value $\mathsf{new}(w_{\text{list}}^{\mathsf{expr}_p}(i))$ (or $\mathtt{nil}$ if it does not exists). Note that during an update step there are at most $\mathit{poly}(Q)$ items, where the fit status is changing. In particular $\mathit{poly}(Q)t_a$ aggregate values are changing in time $\mathit{poly}(Q)t_a$. For every change, we have to update the data structure for $\widetilde{Q}$ on $\widetilde{D}$ in time t_u by doing the updates $\mathsf{delete}R_p(\alpha^i(u_1), \cdots, \alpha^i(u_q), \mathsf{old}(w_{\text{list}}^{\mathsf{expr}_p}(i)), b_1, \ldots, b_p)$, if $\mathsf{old}(w_{\text{list}}^{\mathsf{expr}_p}(i)) \neq \mathtt{nil}$ and $\mathsf{insert}R_p(\alpha^i(u_1), \cdots, \alpha^i(u_q), \mathsf{new}(w_{\text{list}}^{\mathsf{expr}_p}(i)), b_1, \ldots, b_p)$ if $\mathsf{new}(w_{\text{list}}^{\mathsf{expr}_p}(i)) \neq \mathtt{nil}$ for every item i where the $w_{\text{list}}^{\mathsf{expr}_p}(i)$ changes. All the update steps take time $2\,\mathit{poly}(Q)t_u = \mathit{poly}(Q)t_u$. In particular an update step takes time $\mathit{poly}(Q)(t_a + t_u)$. □

Now, we can give a proof for Theorem 3.7.

Proof of Theorem 3.7. Suppose that we receive as input a q-hierarchical CQ with aggregates Q. To initialise the data structure for Q on database D, the algorithm, we obtain from Lemma 6.18(b), computes a corresponding schema $\widetilde{\sigma}$ and a q-hierarchical query $\widetilde{Q}$ (without aggregates) and the q-tree using Lemma 6.8. Using Theorem 3.3, we obtain that there is a data structure for $\widetilde{Q}$ and $\widetilde{\sigma}$-dbs that can be ini-

tialised for the empty database in time $\widetilde{t_i} = O(poly(Q))$ and can be updated in time $\widetilde{t_u} = poly(\widetilde{Q}) \leqslant poly(Q)$. Using Lemma 6.18(c) we obtain an algorithm for Q with initialisation time $t_i = O(poly(Q))$ and update time $t_u = poly(Q)t_a$. Note that the data structure from Theorem 3.3 for $\widetilde{Q}$ on $\widetilde{D}$ is present.

The **enumerate**-routine can be done as follows: Use the **enumerate**-routine from Theorem 3.3(b) for $\widetilde{Q}$ and $\widetilde{D}$. Since $Q(D) = \widetilde{Q}(\widetilde{D})$ (Lemma 6.18(a)), it follows that we enumerate the tuples in $Q(D)$.

The **test**-routine can be done as follows: Use the **test**-routine from Theorem 3.3(a) for $\widetilde{Q}$ and $\widetilde{D}$. Since $Q(D) = \widetilde{Q}(\widetilde{D})$ (Lemma 6.18(a)), it follows that we test upon input of a tuple $\overline{a}$ if $\overline{a} \in Q(D)$.

The difference routine can be done as follows: Use the **diff**-routine from Theorem 3.3(c) for $\widetilde{Q}$ and $\widetilde{D}$. Since $Q(D^-) = \widetilde{Q}(\widetilde{D^-})$ and $Q(D^+) = \widetilde{Q}(\widetilde{D^+})$ (Lemma 6.18(a)), it can enumerate the sets $Q(D^+) \setminus Q(D^-)$ and $Q(D^-) \setminus Q(D^+)$ with delay $poly(Q)$.

The **count**-routine can be done as follows: Use the **count**-routine from Theorem 3.3(a) for $\widetilde{Q}$ and $\widetilde{D}$. Since $Q(D) = \widetilde{Q}(\widetilde{D})$ (Lemma 6.18(a)), it follows that we test on input of a tuple $\overline{a}$ if $\overline{a} \in Q(D)$. □

Using the counting expression in Section 6.5 and Theorem 3.7(a) we can give a proof for Theorem 3.3(d):

Proof of Theorem 3.3(d). On input of a Q and a database D compute the q-tree T_Q and the corresponding query Q_c with the counting expression. This can be computed in $poly(Q)$. Then use Theorem 3.7 to additionally maintain a data structure for Q_c and D. Clearly, since $t_a = O(1)$, we can maintain the data structure with initialisation time $t_i = poly(Q)$ and $t_u = poly(Q)$. Additionally, we store a variable `value` where we store the result $Q_c(D)$ after every update. This can done by using the **enumerate** routine for Q_c and D in Theorem 3.7(a). The **count** routine can be done by outputting `value`. This can be done in $O(1)$. □

7. An application concerning learning polynomial regression models over q-hierarchical queries

In this chapter we show how to use aggregate queries from the previous chapter to efficiently prepare the training data (which is here the result set of a q-hierarchical conjunctive query) for learning a polynomial regression function, i.e., we prove Theorem 3.8 that is restated in the following.

Theorem 3.8. *For every $d \in \mathbb{N}_{\geqslant 1}$ there is a dynamic algorithm that receives a q-hierarchical conjunctive query Q and a σ-db D_0 and computes within $t_i = O(poly(Q)^{2d+1})$ initialisation time and $t_p = poly(Q)^{2d+1} \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)^{2d+1}$ and allows to prepare the learning of the parameters of a polynomial hypothesis function h_Θ of degree d in time $t_l = O(poly(Q)^{2d+1})$ where D is the current database.*

We consider a problem, described by a function $F : \mathbf{dom}^{k-1} \to \mathbf{dom}$ where we receive as input a $(k-1)$-ary tuple $\overline{a}$ of **dom** and map this to $F(\overline{a})$. In the regression task of machine learning we try to approximate the function F from a set

$$\left\{(b^{(1)}, a_1^{(1)}, \ldots, a_{k-1}^{(1)}), \ldots, (b^{(m)}, a_1^{(m)}, \ldots, a_{k-1}^{(m)})\right\}.$$

This ultimate set is called training set and consists of tuples containing the input values $a_1^{(i)}, \ldots, a_{k-1}^{(i)}$ together with their results $b^{(i)}$. The a-values (the input) are called features and the b-value (the output) is called label. In our setting, the training set is taken from a query result $Q(D)$ for a q-hierarchical conjunctive query

$$Q \;=\; \{(y, x_1, \ldots, x_{k-1}, b_1, \ldots, b_\ell) \;:\; \varphi\}$$

where $\{y, x_1, \ldots, x_{k-1}\} = \text{free}(\varphi)$ and $b_1, \ldots, b_\ell \in \mathbf{dom}$. Note that the training set we use is not exactly the query result. We delete in every tuple the components that are constants. Using a feature or a label as constant, i.e., the value in every tuple in the result set is the same, forces that the trained model does not depend on the feature of the constant. This is the reason why we simply ignore constants in this context.

The goal is to define a function h_Θ which is based on the training set and approximates a label for an arbitrary $(k-1)$-ary tuple of features. This is called *hypothesis function*.

Before we start, let us first consider an example that describes an application for machine learning and databases [85].

7. *An application concerning learning polynomial regression models over q-hierarchical queries*

Example 7.1. *Let us assume that our aim is to consult companies, that have web pages, to get a better visibility in search engines by improving their positions. For most companies this is a very interesting question since most users often recognise the first few results of a search engine only. A feasible solution might be the following. We design a database D that consists of a relation R_{Position} of arity 4 and a relation $R_{\mathsf{textlength}}$ of arity 3.*

Some software crawls the search engine on input of a keyword a_{keyword} and inserts for every position a_{Pos} the tuple $(a_{date}, a_{\mathsf{url}}, a_{\mathsf{keyword}}, a_{\mathsf{Pos}})$ into the relation $R_{\mathsf{Position}}(D)$ where a_{url} is the url of the website at position a_{Pos} when requesting the result in the search engine for the keyword a_{keyword} and a_{date} is the current date.

Another software analyses the websites and counts their number of words in the site. The software stores the result in the database by inserting the tuple $(a_{\mathsf{date}}, a_{\mathsf{url}}, b_{\mathsf{length}})$ to $R_{\mathsf{textlength}}(D)$ where b_{length} is the length of the web page, i.e., the number of words in the site, with url a_{url} and a_{date} is the current date.

Let us consider the following q-hierarchical conjunctive query

$$Q_{\mathsf{SEO}} := \left\{(y_{\mathsf{length}}, x_{\mathsf{date}}, x_{\mathsf{url}}, x_{\mathsf{Pos}}) \ : \ \exists x_{\mathsf{keyword}} \left(\begin{array}{c} R_{\mathsf{Position}} x_{date} x_{\mathsf{url}} x_{\mathsf{keyword}} x_{\mathsf{Pos}} \wedge \\ R_{\mathsf{textlength}} x_{\mathsf{date}} x_{\mathsf{url}} y_{\mathsf{length}} \end{array} \right)\right\}.$$

The query result $Q_{\mathsf{SEO}}(D)$ is the set of tuples $(b_{\mathsf{length}}, a_{\mathsf{date}}, a_{\mathsf{url}}, a_{\mathsf{Pos}})$ such that on a_{date} for the website with url a_{url} and length b_{length} there is a keyword such that a_{url} was in position a_{Pos} in the search engine on input of the keyword.

Taking the length as label and the other components as features, we can learn a function F where $F(x_{\mathsf{date}}, x_{\mathsf{url}}, x_{\mathsf{Pos}})$ approximates the length of the website x_{url} on position x_{Pos} on x_{date}. Moreover, we can use the function to check for another position, which text length we need, if we want to get a higher position in the future and we can consult the companies to change the length (In fact, there are a lot of other properties that may change the position in a search engine. For these properties, we can operate in the same manner as for the text length). Another useful application for this function is that we can take the first derivation of the function to receive correlations, such as a correlation between the position and the length.

Regression functions try to minimize the error between the value of the function and the training data. In particular, it might be possible that some tuples in the training data have too much influence in leaning the hypothesis function such that the polynomial does not give a good approximation for most of the training data, e.g. if the text length of the websites on position 1 to 9 and 11 to 20 is between 100 and 200 and the text length of the page in position 10 has length 5000000, the training data for position 10 will approximate the function F such that most of the values of F are over 200. That is the reason why the training data has to be smoothed, i.e. in our example we would delete the text length for the web page in position 10. To smooth the training data, we simply have to delete the corresponding tuples.

For convenience, we assume in this chapter that $\mathbf{dom} = \mathbb{R}$ and we use for the aggregates prod and sum variants where $\mathbf{dom} = \mathbb{R}$, i.e., the binary relations $\circ_{\mathsf{sum}}$ and $\circ_{\mathsf{prod}}$ operate over real numbers.

We consider in Section 7.2 the case that the function h_Θ is a polynomial function. As a warm up, we deal in Section 7.1 with linear hypothesis functions using examples.

7.1. Linear regression

The first part of this section describes the theoretical background of linear regression.

In the linear regression task the hypothesis function is a linear function of the form

$$h_\Theta(\overline{a}) \;:=\; \theta_0 + \theta_1 a_1 + \cdots + \theta_{k-1} a_{k-1}.$$

To approximate the parameters $\Theta \;:=\; (\theta_0, \ldots, \theta_{k-1})$ we minimize the least squares regression error function

$$\xi(\Theta) \;:=\; \frac{1}{2}\sum_{i=1}^{m}\left(h_\Theta(\overline{a}^{(i)}) - y^{(i)}\right)^2 \;=\; \frac{1}{2}\sum_{i=1}^{m}\left(\theta_0 + \sum_{j=1}^{k-1}\theta_j a_j^{(i)} - b^{(i)}\right)^2.$$

Introducing new constants $\theta_k := -1$ and $a_0^{(i)} := 1$ and variables $a_k^{(i)} := b^{(i)}$ for all $i \in [m]$, we can rewrite the error function in a compact way

$$\xi(\Theta) \;=\; \frac{1}{2}\sum_{i=1}^{m}\left(\sum_{j=0}^{k}\theta_j a_j^{(i)}\right)^2.$$

To compute the minimum of $\xi(\Theta)$ we have to compute the jacobi matrix $\mathsf{J}(\xi)$ of $\xi(\Theta)$. For all $\ell \in \{0, \ldots, k-1\}$ the $(1, (\ell+1))$th element of $\mathsf{J}(\xi)$ is:

$$\begin{aligned}\mathsf{J}(\xi)_{1,\ell+1} \;&=\; \frac{\delta}{\delta\theta_\ell}\frac{1}{2}\sum_{i=1}^{m}\left(\sum_{j=0}^{k}\theta_j a_j^{(i)}\right)^2 \;=\; \sum_{i=1}^{m}\left(\sum_{j=0}^{k}\theta_j a_j^{(i)}\right)a_\ell^{(i)} \\ &=\; \sum_{i=1}^{m}\sum_{j=0}^{k}\theta_j a_j^{(i)} a_\ell^{(i)} \;=\; \sum_{j=0}^{k}\sum_{i=1}^{m}\theta_j a_j^{(i)} a_\ell^{(i)} \;=\; \sum_{j=0}^{k}\theta_j \sum_{i=1}^{m} a_j^{(i)} a_\ell^{(i)}.\end{aligned}$$

In the algorithmic task of learning the parameters, we divide the algorithm into two passes. In the first pass, the *forward pass*, one has to compute the sums $\sum_{i=1}^{m} a_j^{(i)} a_\ell^{(i)}$ and in the second pass, the *backward pass*, where one uses a numerical method to learn the parameters Θ.

To learn the parameters Θ we can use a numerical method such as batch gradient descent-method (BGD for short). In the backward pass, using the batch gradient descend method, we set for all $\ell \in \{0, \ldots, k-1\}$ the parameter θ_ℓ to $\theta_\ell - \alpha\mathsf{J}(\xi)_{1,\ell}$. This will be repeated until convergence. The number α is called the learning rate. It represents the step size of the gradient descent. If it is not small enough, $\xi(\Theta)$ may not be smaller after an iteration and if α is too small one need a lot iterations until convergence. The number α may change after an iteration.

7. *An application concerning learning polynomial regression models over q-hierarchical queries*

We focus here on the forward pass. There, a naive algorithm has to compute the sum $\sum_{i=1}^{m} a_j^{(i)} a_\ell^{(i)}$ for all $\ell \in \{0, \ldots, k-1\}$. This takes time $O(m)$. Note that since the training set is taken from the query result, the number m is polynomial in the size of the database. We will show now that if the query is q-hierarchical, we can construct a data structure that can be updated in $O(poly(Q))$ and allows to output the sum in time $O(1)$. For all $u, w \in \mathrm{vars}(Q)$ let

$$\mathsf{Cofactor}[u,w] := \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \beta(u) \cdot \beta(w)$$
$$\text{and} \quad \mathsf{Cofactor}[u,x_0] \;:=\; \mathsf{Cofactor}[x_0,u] \;:=\; \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \beta(u)\ .$$

Note that since the training set is $Q(D)$ it holds that

$$\mathsf{Cofactor}[x_j, x_\ell] = \sum_{i=1}^{m} a_j^{(i)} a_\ell^{(i)} \quad \text{for all } j, \ell \in \{0, \ldots, k\}\ .$$

To compute the cofactor efficiently we use for all $u, w \in \mathrm{vars}(Q)$ the following q-hierarchical queries with aggregates:

$$Q_{u,w} \;:=\; \{(\mathsf{prod}(\mathsf{lre}(u_1), \ldots, \mathsf{lre}(u_\ell)) \;:\; \varphi\}$$

and for all $v \in \mathrm{vars}(Q_{u,w})$ is $\mathsf{lre}(v)$ defined as follows: Let $u_1, \ldots, u_\ell$ be the children of v in T_{free} and for all $i \in [\ell]$ is $e_i := \mathsf{lre}(u_i)$ if $u_i \in \mathrm{path}[u] \cup \mathrm{path}[w]$ and $e_i := c_{u_i}$ otherwise, where c_{u_i} is the aggregate expression for counting from Definition 6.10. For all $m, \ell \in \mathbb{N}$ let $\mathcal{A}_\ell^m = \{(h_\ell^m)_n\}_{n \in \mathbb{N}}$ be the aggregate with

$$(h_\ell^m)_n(\{\!\!\{\langle a_{1,1}, a_{1,2}, \ldots, a_{1,\ell}\rangle, \ldots, \langle a_{n,1}, a_{n,2}, \ldots, a_{n,\ell}\rangle\}\!\!\}) \;:=\; \sum_{i=1}^{n} (a_{i,1})^m \cdot \prod_{k=2}^{\ell} a_{i,k}.$$

Note that the aggregate takes at most $O(1)$ time.

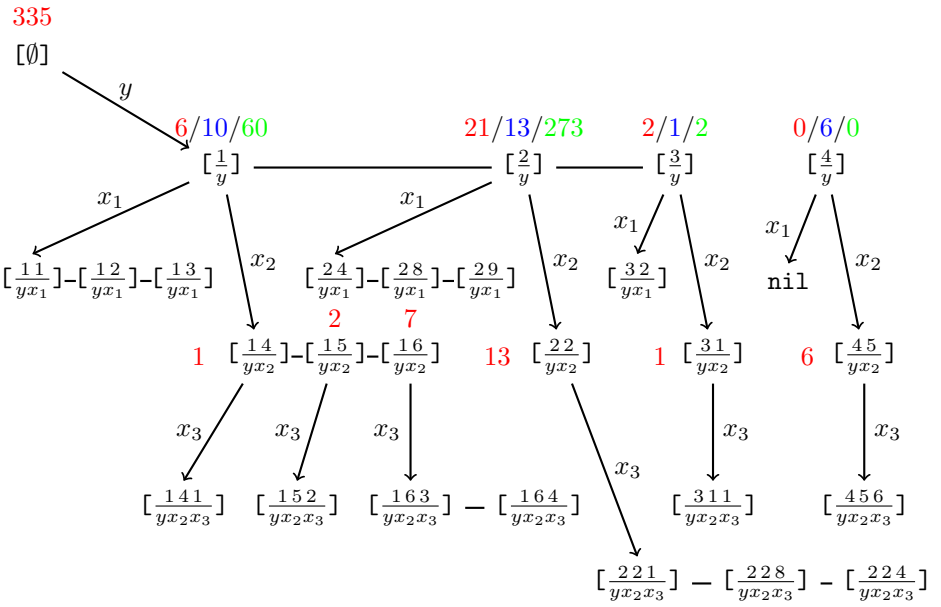
- If $v \in \{u, w\}$ we have $\mathsf{lre}(v) := \mathcal{A}_{\ell+1}^m(\langle v, e_1, \ldots, e_\ell\rangle)$ where $m = 2$ if $u = w$ and $m = 1$ otherwise and
- if $v \notin \{u, w\}$ we have $\mathsf{sum}(\mathsf{prod}(e_1, \ldots, e_\ell))$.

To show that for all $x, y \in \mathrm{vars}(Q)$ is $Q_{x,y}(D) = \mathsf{Cofactor}[x, y]$, one can show that the following claim holds.

Claim 7.2. *Let Q be a q-hierarchical CQ and let D be a σ-db. For all items i in the data structure for Q on D, with $v^i \in \mathrm{path}[u] \cup \mathrm{path}[w]$ holds for all $p \in (\mathrm{path}[u] \cup \mathrm{path}[w]) \cap child(v^i)$ the following is satisfied.*

- *If $p \in \mathrm{path}[u] \setminus \mathrm{path}[w]$ then $[\![\mathsf{lre}(p)]\!]^{(D,\alpha^i)} = \sum_{\beta \in \widetilde{\mathcal{E}}^i} \beta(u)$.*

Figure 7.1.: Illustration of Q_{x_1,x_3} on the database D of Example 4.4.



- *If* $p \in \mathrm{path}[w] \setminus \mathrm{path}[u]$ *then* $[\![\mathsf{Ire}(p)]\!]^{(D,\alpha^i)} = \sum_{\beta \in \widetilde{\mathcal{E}}^i} \beta(w)$.
- *If* $p \in \mathrm{path}[u] \cap \mathrm{path}[w]$ *then* $[\![\mathsf{Ire}(p)]\!]^{(D,\alpha^i)} = \sum_{\beta \in \widetilde{\mathcal{E}}^i} \beta(u) \cdot \beta(w)$.

Note that $Q_{u,w}(D) = \mathsf{Cofactor}[u,w]$ follows from Claim 7.2 for $i = [\emptyset]$ since $v_{root} \in \mathrm{path}[u) \cap \mathrm{path}[w)$.

A proof of Claim 7.2 can be obtained by the proof of Lemma 7.3 for polynomial function by considering the special case that $d = 1$.

To get an idea how it works let us consider the database and the query from Example 4.4. For the example, we are interested in $\mathsf{Cofactor}[x_1,x_3]$ and $\mathsf{Cofactor}[x_3,x_3]$. The corresponding queries are the following.

$$Q_{x_1,x_3} = \{(\mathsf{prod}(\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_1\rangle), \mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_3\rangle)))))))) \ : \ \varphi\}$$
$$Q_{x_3,x_3} = \{(\mathsf{prod}(\mathsf{sum}(\mathsf{prod}(\mathsf{count}(x_1), \mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^2(\langle x_3\rangle)))))))) \ : \ \varphi\}$$
$$\text{where} \qquad \varphi \ := \ Eyx_1 \ \wedge \ Fyx_2x_3 \ \wedge \ Gyx_2x_3$$

See Figure 7.1 for an illustration of Q_{x_1,x_3} on the database D_0 of Example 4.4. A red number on the left or above an item i with $v^i = x_2$ represents the number $w_{\mathrm{list}}^{\mathcal{A}_1^1(\langle x_3\rangle)}(i)$, i.e., the number of the sum over $\iota \in \mathcal{L}_{x_3}^i$ of their constants a^ι. Since x_2 has only one child it follows that $w_{\mathrm{item}}^{\mathsf{prod}(\mathcal{A}_1^1(\langle x_3\rangle))}(i) = w_{\mathrm{list}}^{\mathcal{A}_1^1(\langle x_3\rangle)}(i)$ for all items i with $v^i = x_2$. The three numbers $n_1/n_2/n_3$ above an item i with $v^i = y$ represents that $n_1 =$

7. *An application concerning learning polynomial regression models over q-hierarchical queries*

y	x_1	x_2	x_3		$x_1 \cdot x_3$		$(x_3)^2$
1	1	4	1		1		1
1	1	5	2		2		4
1	1	6	3		3		9
1	1	6	4		4		16
1	2	4	1		2		1
1	2	5	2		4		4
1	2	6	3		6		9
1	2	6	4		8		16
1	3	4	1		3		1
1	3	5	2		6		4
1	3	6	3		9		9
1	3	6	4		12		16
2	4	2	1		4		1
2	4	2	8		32		64
2	4	2	4		16		16
2	8	2	1		8		1
2	8	2	8		64		64
2	8	2	4		32		16
2	9	2	1		9		1
2	9	2	8		72		64
2	9	2	4		36		16
3	2	3	1		2		1
				$\sum =$	**335**	$\sum =$	**334**

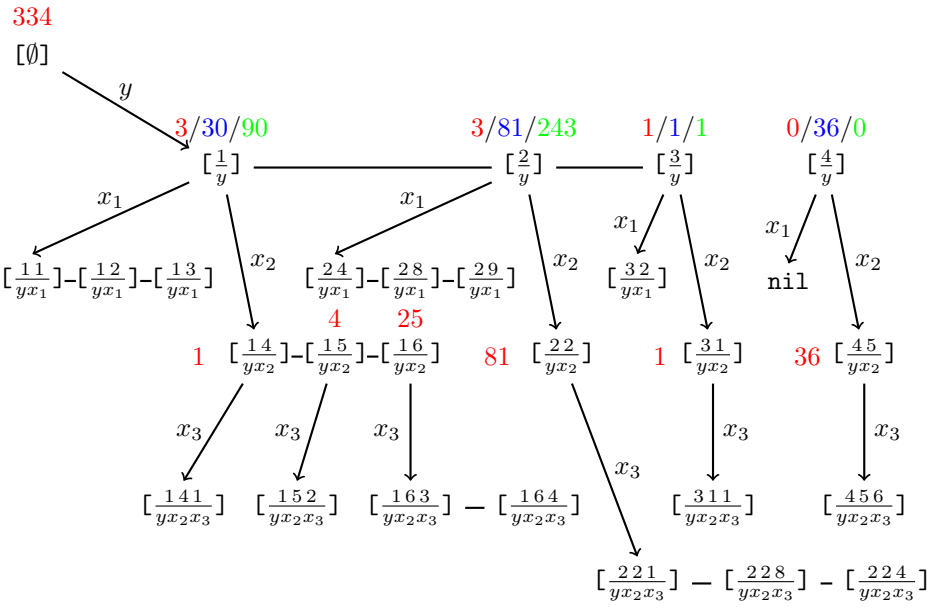
Table 7.1.: Enumeration of $Q(D_0)$ from Example 4.4 together with the product $x_1 \cdot x_3$ and $x_3 \cdot x_3$ for every result tuple and the sum over all result tuples of these products.

$w_{\text{list}}^{\mathcal{A}_1^1(\langle x_1 \rangle)}(i)$, i.e., n_1 is the number of the sum of a^ι over $\iota \in \mathcal{L}_{x_1}^i$, the number $n_2 = w_{\text{list}}^{\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_3 \rangle)))}(i)$, i.e., the sum over $\iota \in \mathcal{L}_{x_2}^i$ of their $w_{\text{item}}^{\mathsf{prod}(\mathcal{A}_1^1(\langle x_3 \rangle))}(\iota)$ values. The number n_3 is $w_{\text{list}}^{\mathsf{prod}(\mathcal{A}_1^1(\langle x_1 \rangle),\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_3 \rangle))))}(i)$, i.e., the product of n_1 and n_2. The red number above the start item depicts $w_{\text{list}}^{\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_1 \rangle),\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_3 \rangle))))}(\texttt{[}\emptyset\texttt{]})$. Note that since the node v_{root} has only one child in the q-tree of Q_{x_1,x_2}, the number is also the query result $Q_{x_1,x_2}(D)$.

See Table 7.1 for the query result of $Q(D)$ and the product $\beta(x_1) \cdot \beta(x_3)$ and $(\beta(x_3))^2$ for every tuple in the result set. These products are given in the line of the corresponding tuple. At the bottom of the columns of $x_1 \cdot x_3$ and $(x_3)^2$ the sum over all products is given, which is equal to $\sum_{\beta \in \widetilde{\mathcal{E}}^{\texttt{[}\emptyset\texttt{]}}} \beta(x_1) \cdot \beta(x_3)$ (resp. $\sum_{\beta \in \widetilde{\mathcal{E}}^{\texttt{[}\emptyset\texttt{]}}} \beta(x_3)^2$)

Let us consider the item $i = \texttt{[}\frac{1}{y}\texttt{]}$. Recall the value n_3 of i from the previous

Figure 7.2.: Illustration of Q_{x_3,x_3} on the database D of Example 4.4.



description:

$$\begin{aligned} n_3 &= (1+2+3)(1+2+(3+4)) \\ &= 1\cdot 1 + 1\cdot 2 + 1\cdot 3 + 1\cdot 4 + 2\cdot 1 + 2\cdot 2 + 2\cdot 3 + 2\cdot 4 + 3\cdot 1 + 3\cdot 2 + 3\cdot 3 + 3\cdot 4 \end{aligned}$$

Compare these values with those in Table 7.1, we see, that n_3 is the sum over all assignments $\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}$ with $\beta(y) = 1$ of $\beta(x_1)\cdot\beta(x_3)$. The number n_3 is equal to $\sum_{\beta\in\widetilde{\mathcal{E}}^{[\frac{1}{y}]}} \beta(x_1)\cdot\beta(x_3)$ since the decomposition lemma (Lemma 4.12) states that each assignment in $\widetilde{\mathcal{E}}^{[\frac{1}{y}]}$ is the union of assignments β_1 and β_2 where for β_1 there is an item $\iota_1 \in \mathcal{L}_{x_1}^{[\frac{1}{y}]}$ such that $\beta_1 \in \widetilde{\mathcal{E}}^{\iota_1}$ and for β_2 there is an item $\iota_2 \in \mathcal{L}_{x_2}^{[\frac{1}{y}]}$ such that $\beta_2 \in \widetilde{\mathcal{E}}^{\iota_2}$ and thus $\sum_{\beta\in\widetilde{\mathcal{E}}^{[\frac{1}{y}]}} \beta(x_1)\cdot\beta(x_3) = \sum_{\iota_1\in\mathcal{L}_{x_1}^{[\frac{1}{y}]}} \sum_{\beta_1\in\widetilde{\mathcal{E}}^{\iota_1}} \beta_1(x_1) \sum_{\iota_2\in\mathcal{L}_{x_2}^{[\frac{1}{y}]}} \sum_{\beta_1\in\widetilde{\mathcal{E}}^{\iota_2}} \beta_2(x_2) = n_1\cdot n_2$.

The same can be shown for the items $[\frac{2}{y}]$ and $[\frac{3}{y}]$. Since

$$w_{\text{list}}^{\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_1\rangle),\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^1(\langle x_3\rangle))))}([\emptyset])$$

is the sum of all the n_3 values of $[\frac{1}{y}]$, $[\frac{2}{y}]$ and $[\frac{3}{y}]$ it follows that the query result is $\sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \beta(x_1)\cdot\beta(x_3)$.

Let us now consider the query Q_{x_3,x_3} and the corresponding Figure 7.2.

A red number on the left or above an item i with $v^i = x_2$ depicts the number $w_{\text{list}}^{\mathcal{A}_1^2(\langle x_3\rangle)}(i)$, i.e., the number of the sum over $\iota \in \mathcal{L}_{x_3}^i$ of the square of their constants

7. *An application concerning learning polynomial regression models over q-hierarchical queries*

a^ι. Note that since x_2 has only one child it follows that $w_{\text{item}}^{\mathsf{prod}(\mathcal{A}_1^2(\langle x_3\rangle))}(i) = w_{\text{list}}^{\mathcal{A}_1^2(\langle x_3\rangle)}(i)$ for all items i with $v^i = x_2$. The three numbers $n_1/n_2/n_3$ above an item i with $v^i = y$ depict that $n_1 = w_{\text{list}}^{\mathsf{count}(x_1)}(i)$, i.e., is the number of item in the $\mathcal{L}_{x_1}^i$-list, the number $n_2 = w_{\text{list}}^{\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^2(\langle x_3\rangle)))}(i)$, i.e., the sum over $\iota \in \mathcal{L}_{x_2}^i$ of their $w_{\text{item}}^{\mathsf{prod}(\mathcal{A}_1^2(\langle x_3\rangle))}(\iota)$ values. The number n_3 is $w_{\text{list}}^{\mathsf{prod}(\mathsf{count}(x_1),\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^2(\langle x_3\rangle))))}(i)$, i.e., the product of n_1 and n_2. The red number above the start item depicts $w_{\text{list}}^{\mathsf{sum}(\mathsf{prod}(\mathsf{count}(x_1),\mathsf{sum}(\mathsf{prod}(\mathcal{A}_1^2(\langle x_3\rangle)))))}(\texttt{[}\emptyset\texttt{]})$. Note that since the node v_{root} has only one child in the q-tree of Q_{x_3,x_3}, the number is equivalent to the query result $Q_{x_3,x_3}(D)$.

Let us consider the item $i = \texttt{[}\frac{1}{y}\texttt{]}$. Recall that the n_3-value of i is computed the following way:

$$n_3 \;:=\; 3 \cdot (1 + 4 + (9 + 16))$$

Comparing this with Table 7.1 we see, that the values $(x_3)^2$ will be repeated 3 times. This depends on the fact that the values for x_3 will be repeated since every item in all assignments $\beta_1 \in \widetilde{\mathcal{E}}^{\iota_1}$ for every $\iota_1 \in \mathcal{L}_{x_1}^i$ has to be compared with $\beta_2 \in \widetilde{\mathcal{E}}^{\iota_2}$ for every $\iota_2 \in \mathcal{L}_{x_2}^i$ to receive an assignment in $\widetilde{\mathcal{E}}^i$. In particular, every $\beta_2 \in \widetilde{\mathcal{E}}^{\iota_2}$ for every $\iota_2 \in \mathcal{L}_{x_2}^i$ will be enumerated three times since there are three assignments in $\left|\bigcup_{\iota_1 \in \mathcal{L}_{x_1}^{\texttt{[}\frac{1}{y}\texttt{]}}} \widetilde{\mathcal{E}}^{\iota_1}\right|$. This is the number given by n_1. Analogously, this can be shown for the items $\texttt{[}\frac{2}{y}\texttt{]}$ and $\texttt{[}\frac{3}{y}\texttt{]}$ and it is straightforward to see that their sums are equal to $\sum_{\beta \in \widetilde{\mathcal{E}}^{\texttt{[}\emptyset\texttt{]}}} \beta(x_3)^2$.

7.2. Polynomial regression

In this section we enrich the idea for maintaining linear regression to polynomial regression, i.e., we show the result for polynomial regression of degree d. The hypothesis function for a polynomial of degree d is the following.

$$h_\Theta(a_1, \cdots, a_k) = \sum_{j_1=0}^{k} \sum_{j_2=j_1}^{k} \cdots \sum_{j_d=j_{d-1}}^{k} \theta_{(j_1,\cdots,j_d)} \cdot a_{j_1} \cdot \cdots \cdot a_{j_d}$$

where $a_0 := 1$ and $\Theta = (\theta_i)_{i \in \mathcal{I}_d}$ such that $\mathcal{I}_d$ denotes the index set for d which is defined as $\mathcal{I}_d := \{(j_1, \cdots, j_d) : j_1, \ldots, j_d \in \{0, \ldots, k\}, j_1 \leqslant \cdots \leqslant j_d\}$. Using the least squares error function we receive

$$\begin{aligned}\xi(\Theta) &:= \frac{1}{2} \sum_{i=1}^{m} \left(h_\Theta(\overline{a}^{(i)}) - b^{(i)}\right)^2 \\ &= \frac{1}{2} \sum_{i=1}^{m} \left(\sum_{(j_1,\ldots,j_d) \in \mathcal{I}_d} \theta_{(j_1,\cdots,j_d)} \cdot a_{j_1}^{(i)} \cdot \cdots \cdot a_{j_d}^{(i)} - b^{(i)} \right)^2\end{aligned}$$

To compute the minimum of the error function, we consider the Jacobi matrix $\mathsf{J}(\xi)$ of $\xi(\Theta)$. The $(1,(\ell_1,\cdots,\ell_d))$th component of the Jacobi matrix is

$$\begin{aligned}
\mathsf{J}(\xi)_{1,(\ell_1,\cdots,\ell_d)} &= \sum_{i=1}^{m} \left(\sum_{(j_1,\ldots,j_d)\in\mathcal{I}_d} \theta_{(j_1,\cdots,j_d)} \cdot a_{j_1}^{(i)} \cdot \;\cdots\; \cdot a_{j_d}^{(i)} - b^{(i)} \right) \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)} \\
&= \sum_{i=1}^{m} \sum_{(j_1,\ldots,j_d)\in\mathcal{I}_d} \theta_{(j_1,\cdots,j_d)} \cdot a_{j_1}^{(i)} \cdot \;\cdots\; \cdot a_{j_d}^{(i)} \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)} \\
&\quad - \sum_{i=1}^{m} b^{(i)} \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)} \\
&= \sum_{(j_1,\ldots,j_d)\in\mathcal{I}_d} \theta_{(j_1,\cdots,j_d)} \sum_{i=1}^{m} a_{j_1}^{(i)} \cdot \;\cdots\; \cdot a_{j_d}^{(i)} \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)} \\
&\quad - \sum_{i=1}^{m} b^{(i)} \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)}
\end{aligned}$$

To learn the parameters in the backward pass, one can use the batch gradient descent method (see Algorithm 10).

Algorithm 10 Learn the parameters Θ using the batch gradient descent method on polynomial regression.

1: **repeat**
2: **for all** $(\ell_1,\cdots,\ell_d)\in\mathcal{I}_d$ **do**
3: Set $\theta_{(\ell_1,\cdots,\ell_d)}$ to $\theta_{(\ell_1,\cdots,\ell_d)} - \alpha\mathsf{J}(\xi)_{1,(\ell_1,\cdots,\ell_d)}$
4: **until** convergence

The number α is once again the learning rate. We discuss how to efficiently maintain the sums of the form $\sum_{i=1}^{m} a_{j_1}^{(i)} \cdot \;\cdots\; \cdot a_{j_d}^{(i)} \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)}$ under updates. Since the training set is taken from the query result, we obtain that

$$\sum_{i=1}^{m} a_{j_1}^{(i)} \cdot \;\cdots\; \cdot a_{j_d}^{(i)} \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)} = \sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{d} \beta(x_{j_p}) \cdot \beta(x_{\ell_p})$$

and

$$\sum_{i=1}^{m} b^{(i)} \cdot a_{\ell_1}^{(i)} \cdot \;\cdots\; \cdot a_{\ell_d}^{(i)} = \sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \beta(y) \prod_{p=1}^{d} \beta(x_{\ell_p})$$

Here, we set $\beta(x_0) := 1$ for all $\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}$. The following lemma shows that these sums can be maintained under updates.

7. *An application concerning learning polynomial regression models over q-hierarchical queries*

Lemma 7.3. *There is a dynamic algorithm that receives a q-hierarchical CQ Q and a σ-db D_0 and computes within $t_i = poly(Q)^{2d}$ initialisation time, and $t_p = poly(Q)^{2d}\|D\|$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows upon input of a tuple $(w_1, \ldots, w_{\widetilde{d}}) \in \mathrm{vars}(Q)^{\widetilde{d}}$ for $\widetilde{d} \leqslant 2d$ the value*

$$\sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{\widetilde{d}} \beta(w_p)$$

in time $O(1)$.

Proof. To handle these sums under updates, we define for every $\widetilde{d} \leqslant 2d$ and every $(w_1, \cdots, w_{\widetilde{d}}) \in \mathrm{vars}(Q)^{\widetilde{d}}$ a query $Q_{(w_1,\cdots,w_{\widetilde{d}})}$ such that

$$Q_{(w_1,\cdots,w_{\widetilde{d}})}(D) \;=\; \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{\widetilde{d}} \beta(w_p).$$

In the following we show now how to define a query $Q_{\overline{w}}$ for an arbitrary tuple $\overline{w} = (w_1, \ldots, w_{\widetilde{d}})$. Let T_{free} be the subgraph of the q-tree of Q induced on $\mathrm{free}(Q)$ and let $\widetilde{T}_{\overline{w}}$ be the subgraph of T_{free} induced on $\bigcup_{i=1}^{\widetilde{d}} \mathrm{path}[w_i]$. For all $v \in \mathrm{vars}(Q)$ we define $\#v$ as the number of appearance of v in the tuple $\overline{w}$.

Definition 7.4. *The aggregate expression $\mathsf{reg}_{\overline{w}}$ is defined as follows. For all $v \in V(\widetilde{T}_{\overline{w}})$ let $u_1, \cdots, u_s$ be the children of v in T_{free} and $e_i := \mathsf{reg}_{\overline{w}}(u_i)$ if $u_i \in V(\widetilde{T}_{\overline{w}})$ and $e_i := c_{u_i}$ otherwise where c_{u_i} is the counting expression from Definition 6.10. Then,*

- *if $v \in \{w_1, \ldots, w_d\}$ then $\mathsf{reg}_{\overline{w}}(v) := \mathcal{A}_{s+1}^{\#v}(\langle v, e_1, \cdots, e_s\rangle)$ and*
- *if $v \notin \{w_1, \ldots, w_d\}$ then $\mathsf{reg}_{\overline{w}}(v) := \mathsf{sum}(\mathsf{prod}(e_1, \cdots, e_s))$.*

For any q-hierarchical conjunctive query $Q = \{(x_1, \ldots, x_k) : \varphi\}$ the query $Q_{\overline{w}} = \{(\mathsf{prod}(e_1, \ldots, e_s)) : \varphi\}$ where $z_1, \ldots, z_s$ are the children of the root in the q-tree of Q and $e_j := \mathsf{reg}_{\overline{w}}(z_j)$ if $z_j \in V(\widetilde{T}_{\overline{w}})$ and $e_j := c_{z_j}$ otherwise.

Our aim is to show that

$$Q_{(w_1,\cdots,w_{\widetilde{d}})}(D) \;=\; \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{\widetilde{d}} \beta(w_p).$$

The following notation and the following claim will be convenient for the proof.

For all $v \in V(\widetilde{T}_{\overline{w}})$ let

$$M_v \;:=\; \mathsf{succ}(v) \cap V(\widetilde{T}_{\overline{w}}) \quad \text{and} \quad W_v \;:=\; M_v \cap \{w_1, \ldots, w_d\}\,.$$

Claim 7.5. *For all items i with $v^i \in V(\widetilde{T}_{\overline{w}})$ we have*

$$[\![\mathsf{reg}_{\overline{w}}(u)]\!]^{(D,\alpha^i)} = \sum_{\iota \in \mathcal{L}_u^i} \sum_{\beta \in \widetilde{\mathcal{E}}^\iota} \prod_{z \in W_u} \beta(z)^{\#z}$$

for all $u \in M_{v^i} \cap \mathit{child}(v^i)$.

We show this claim by induction of the height of an item in $\widetilde{T}_{\overline{w}}$.

For the induction base let us consider an item i where v^i has height 1 in $\widetilde{T}_{\overline{w}}$. Then, for all $u \in M_{v^i}$ it holds that $u \in \{w_1, \ldots, w_d\}$. Let $z_1, \cdots, z_s$ where $s \geqslant 0$ be the children of v^i in T_{free}. Note that $e_j = c_{z_j}$ for all $j \in [s]$ since $z_j \notin V(\widetilde{T}_{\overline{w}})$.

$$\begin{aligned}
[\![\mathsf{reg}_{\overline{w}}(u)]\!]^{(D,\alpha^i)} &= \sum_{\iota \in \mathcal{L}_u^i} (a^\iota)^{\#u} \cdot \prod_{j \in [s]} [\![c_{v_j}]\!]^{(D,\alpha^\iota)} \\
&\overset{(\star)}{=} \sum_{\iota \in \mathcal{L}_u^i} (a^\iota)^{\#u} \cdot \prod_{j \in [s]} \left| \bigcup_{\hat{\iota} \in \mathcal{L}_{v_j}^\iota} \widetilde{\mathcal{E}}^{\hat{\iota}} \right| \overset{(\star\star)}{=} \sum_{\iota \in \mathcal{L}_u^i} (a^\iota)^{\#u} \cdot \left| \widetilde{\mathcal{E}}^\iota \right| \\
&\overset{(\star\star\star)}{=} \sum_{\iota \in \mathcal{L}_u^i} \sum_{\beta \in \widetilde{\mathcal{E}}^\iota} \beta(u)^{\#u} \overset{(\star\star\star\star)}{=} \sum_{\iota \in \mathcal{L}_u^i} \sum_{\beta \in \widetilde{\mathcal{E}}^\iota} \prod_{z \in W_u} \beta(z)^{\#z}
\end{aligned}$$

Equation $(\star)$ follows from Claim 6.13, and Equation $(\star\star)$ from Claim 6.12, and Equation $(\star\star\star)$ from the fact that $\left|\widetilde{\mathcal{E}}^\iota\right| = \sum_{\beta \in \widetilde{\mathcal{E}}^\iota} 1$ and for all $\iota \in \mathcal{L}_u^i$ and for all $\beta \in \widetilde{\mathcal{E}}^\iota$ we have $\beta(u) = \alpha^\iota$. Equation $(\star\star\star\star)$ follows from the fact that $W_u = \{u\}$.

For the inductive step we consider an item i where v^i has height $h > 1$ in $\widetilde{T}_{\overline{w}}$. Let $u \in M_{v^i}$ be arbitrary and let $v_1, \ldots, v_s$ be the children of u in T. Let us consider the case that $u \in \{w_1, \ldots, w_d\}$. Then, it follows that

$$\begin{aligned}
[\![\mathsf{reg}_{\overline{w}}(u)]\!]^{(D,\alpha^i)} &= \sum_{\iota \in \mathcal{L}_u^i} (a^\iota)^{\#u} \cdot \prod_{\substack{j \in [s] \\ v_j \notin M_u}} [\![c_{v_j}]\!]^{(D,\alpha^\iota)} \cdot \prod_{\substack{j \in [s] \\ v_j \in M_u}} [\![\mathsf{reg}_{\overline{w}}(v_j)]\!]^{(D,\alpha^\iota)} \\
&\overset{(\star)}{=} \sum_{\iota \in \mathcal{L}_u^i} (a^\iota)^{\#u} \cdot \prod_{\substack{j \in [s] \\ v_j \notin M_u}} \left| \bigcup_{\iota \in \mathcal{L}_{v_j}^\iota} \widetilde{\mathcal{E}}^\iota \right| \cdot \prod_{\substack{j \in [s] \\ v_j \in M_u}} \sum_{\hat{\iota} \in \mathcal{L}_{v_j}^\iota} \sum_{\gamma \in \widetilde{\mathcal{E}}^{\hat{\iota}}} \prod_{z \in W_{v_j}} \gamma(z)^{\#z} \\
&\overset{(\star\star)}{=} \sum_{\iota \in \mathcal{L}_u^i} \sum_{\beta \in \widetilde{\mathcal{E}}^\iota} \prod_{z \in W_u} \beta(z)^{\#z}
\end{aligned}$$

Equation $(\star)$ follows from Claim 6.13 and the induction hypothesis.

Equation $(\star\star)$ follows from the following fact. From the decomposition lemma (Lemma 4.12) we know that for every $\iota \in \mathcal{L}_u^i$ every assignment $\beta \in \widetilde{\mathcal{E}}^\iota$ can be decomposed into $\beta = \beta_u \cup \beta_1 \cup \cdots \cup \beta_s$ such that there is a $\iota_j \in \mathcal{L}_{v_j}^\iota$ where $\beta_j \in \widetilde{\mathcal{E}}^{\iota_j}$ for

all $j \in [s]$. Therefore, it follows

$$\begin{aligned}
\sum_{\iota\in\mathcal{L}_u^i}\sum_{\beta\in\widetilde{\mathcal{E}}^\iota}\prod_{z\in W_u}\beta(z)^{\#z} &\overset{4.12}{=} \sum_{\iota\in\mathcal{L}_u^i}\sum_{\iota_1\in\mathcal{L}_{v_1}^\iota}\sum_{\beta_1\in\widetilde{\mathcal{E}}^{\iota_1}}\cdots\sum_{\iota_s\in\mathcal{L}_{v_s}^\iota}\sum_{\beta_s\in\widetilde{\mathcal{E}}^{\iota_s}}\prod_{z\in W_u}\beta(z)^{\#z} \\
&\qquad \text{where } \beta := \alpha^\iota \cup \beta_1 \cup \cdots \cup \beta_s \\
&= \sum_{\iota\in\mathcal{L}_u^i}\beta(u)^{\#u}\sum_{\iota_1\in\mathcal{L}_{v_1}^\iota}\sum_{\beta_1\in\widetilde{\mathcal{E}}^{\iota_1}}\prod_{z\in W_u\cap\mathsf{succ}(v_1)}\beta(z)^{\#z}\cdots \\
&\qquad \sum_{\iota_s\in\mathcal{L}_{v_s}^\iota}\sum_{\beta_s\in\widetilde{\mathcal{E}}^{\iota_s}}\prod_{z\in W_u\cap\mathsf{succ}(v_s)}\beta(z)^{\#z} \\
&= \sum_{\iota\in\mathcal{L}_u^i}(a^\iota)^{\#u}\prod_{\substack{j\in[s]\\ v_j\notin M_u}}\sum_{\iota_j\in\mathcal{L}_{v_j}^\iota}\sum_{\beta_j\in\widetilde{\mathcal{E}}^{\iota_j}}\prod_{z\in W_{v_j}}\beta_j(z)^{\#z}\cdot \\
&\qquad \prod_{\substack{j\in[s]\\ v_j\in M_u}}\sum_{\iota_s\in\mathcal{L}_{v_s}^\iota}\sum_{\beta_s\in\widetilde{\mathcal{E}}^{\iota_s}}\prod_{z\in W_{v_j}}\beta_j(z)^{\#z}
\end{aligned} \tag{7.1}$$

The last equation holds since $\beta_j \subseteq \beta$ and $\mathrm{dom}(\beta_j) \supseteq W_{v_j}$ and $W_{v_j} = (W_u \cap \mathsf{succ}(v_j))$. For all $j \in [s]$ with $v_j \notin M_u$ it follows that the product $\prod_{z\in W_u\cap\mathsf{succ}(v_j)}\beta(z)^{\#z}$ is empty since $v_j \notin M_u$ and thus $W_u \cap \mathsf{succ}(v_j) = \emptyset$ and therefore we obtain

$$\begin{aligned}
\sum_{\iota_j\in\mathcal{L}_{v_j}^\iota}\sum_{\beta_j\in\widetilde{\mathcal{E}}^{\iota_j}}\prod_{z\in W_{v_j}}\beta(z)^{\#z} &= \sum_{\iota_j\in\mathcal{L}_{v_j}^\iota}\sum_{\beta_j\in\widetilde{\mathcal{E}}^{\iota_j}}1 = \sum_{\iota_j\in\mathcal{L}_{v_j}^\iota}\left|\widetilde{\mathcal{E}}^{\iota_j}\right| \\
&= \left|\bigcup_{\iota_j\in\mathcal{L}_{v_j}^\iota}\widetilde{\mathcal{E}}^{\iota_j}\right|
\end{aligned} \tag{7.2}$$

The last equation follows from the fact that $\widetilde{\mathcal{E}}^{\iota_1} \cap \widetilde{\mathcal{E}}^{\iota_2} = \emptyset$ for $\iota_1, \iota_2 \in \mathcal{L}_u^i$ with $\iota_1 \neq \iota_2$. Comparing equations (7.1) and (7.2) we obtain that

$$\begin{aligned}
&\sum_{\iota\in\mathcal{L}_u^i}\sum_{\beta\in\widetilde{\mathcal{E}}^\iota}\prod_{z\in W_u}\beta(z)^{\#z} \\
&\quad = \sum_{\iota\in\mathcal{L}_u^i}(a^\iota)^{\#u}\prod_{\substack{j\in[s]\\ v_j\notin M_u}}\left|\bigcup_{\iota_j\in\mathcal{L}_{v_j}^\iota}\widetilde{\mathcal{E}}^{\iota_j}\right|\cdot\prod_{\substack{j\in[s]\\ v_j\in M_u}}\sum_{\iota_s\in\mathcal{L}_{v_s}^\iota}\sum_{\beta_s\in\widetilde{\mathcal{E}}^{\iota_s}}\prod_{z\in W_{v_j}}\beta_j(z)^{\#z}
\end{aligned}$$

This concludes the proof of the correctness of $(\star\star)$. Let us now consider the last case that $u \notin \{w_1, \ldots, w_{\widetilde{d}}\}$. Then,

$$[\![\mathsf{reg}_{\overline{w}}(u)]\!]^{(D,\alpha^i)} = \sum_{\iota\in\mathcal{L}_u^i}\prod_{\substack{j\in[s]\\ v_j\notin M_u}}[\![c_{v_j}]\!]^{(D,\alpha^\iota)}\cdot\prod_{\substack{j\in[s]\\ v_j\in M_u}}[\![\mathsf{reg}_{\overline{w}}(v_j)]\!]^{(D,\alpha^\iota)}$$

The proof for that case can be taken verbatim from the case where $u \in \{w_1, \dots, w_{\widetilde{d}}\}$ and u with the following modification. Remove every appearance of $(a^\iota)^{\#u}$ and $(\beta(u))^{\#u}$. This concludes the proof of Claim 7.5.

It remain to show that

$$Q_{(w_1,\cdots,w_{\widetilde{d}})}(D) \;=\; \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{\widetilde{d}} \beta(w_p).$$

By definition it follows:

$$\begin{aligned} Q_{(w_1,\cdots,w_{\widetilde{d}})}(D) \;&=\; [\![\mathsf{prod}(e_1,\dots,e_s)]\!]^{(D,\emptyset)} \\ &=\; \prod_{\substack{j\in[s]\\ y_j \notin V(\widetilde{T}_{\overline{w}})}} [\![c_{z_j}]\!]^{(D,\emptyset)} \cdot \prod_{\substack{j\in[s]\\ z_j \in V(\widetilde{T}_{\overline{w}})}} [\![\mathsf{reg}_{\overline{w}}(z_j)]\!]^{(D,\emptyset)} \\ &\overset{\text{Claim 7.5}}{=} \prod_{\substack{j\in[s]\\ y_j \notin V(\widetilde{T}_{\overline{w}})}} \left| \bigcup_{\iota_j \in \mathcal{L}^{[\emptyset]}_{z_j}} \widetilde{\mathcal{E}}^{\iota_j} \right| \cdot \prod_{\substack{j\in[s]\\ z_j \in V(\widetilde{T}_{\overline{w}})}} \sum_{\iota \in \mathcal{L}^{[\emptyset]}_{y_j}} \sum_{\beta \in \widetilde{\mathcal{E}}^{\iota}} \prod_{q \in W_{z_j}} \beta(q)^{\#q} \end{aligned} \tag{7.3}$$

Furthermore, we have

$$\begin{aligned} \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{\widetilde{d}} \beta(w_p) &\overset{(\star)}{=} \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{q \in W_{v_{root}}} \beta(q)^{\#q} \\ &\overset{(\star\star)}{=} \sum_{\iota_1 \in \mathcal{L}^{[\emptyset]}_{y_1}} \sum_{\beta_1 \in \widetilde{\mathcal{E}}^{\iota_1}} \cdots \sum_{\iota_s \in \mathcal{L}^{[\emptyset]}_{z_s}} \sum_{\beta_s \in \widetilde{\mathcal{E}}^{\iota_s}} \prod_{q \in W_{v_{\text{root}}}} \beta(q)^{\#q} \\ &\qquad\qquad \text{where } \beta \;:=\; \beta_1 \cup \cdots \cup \beta_s \\ &\overset{(\star\star\star)}{=} \prod_{\substack{j\in[s]\\ z_j \notin V(\widetilde{T}_{\overline{w}})}} \sum_{\iota_j \in \mathcal{L}^{[\emptyset]}_{z_j}} \sum_{\beta_j \in \widetilde{\mathcal{E}}^{\iota_j}} \prod_{q \in W_{z_j}} \beta_j(q)^{\#q} \cdot \prod_{\substack{j\in[s]\\ z_j \in V(\widetilde{T}_{\overline{w}})}} \sum_{\iota_j \in \mathcal{L}^{[\emptyset]}_{z_j}} \sum_{\beta_j \in \widetilde{\mathcal{E}}^{\iota_j}} \prod_{q \in W_{w_j}} \beta_j(q)^{\#q} \\ &\overset{(\star\star\star\star)}{=} \prod_{\substack{j\in[s]\\ z_j \notin V(\widetilde{T}_{\overline{w}})}} \left| \bigcup_{\iota_j \in \mathcal{L}^{\iota}_{z_j}} \widetilde{\mathcal{E}}^{\iota_j} \right| \cdot \prod_{\substack{j\in[s]\\ z_j \in V(\widetilde{T}_{\overline{w}})}} \sum_{\iota_j \in \mathcal{L}^{[\emptyset]}_{z_j}} \sum_{\beta_j \in \widetilde{\mathcal{E}}^{\iota_j}} \prod_{q \in W_{z_j}} \beta_j(q)^{\#q} \\ &\overset{(7.3)}{=} Q_{(w_1,\cdots,w_{\widetilde{d}})}(D) \end{aligned}$$

For Equation $(\star)$ note that every element $z \in W_{v_{root}}$ occurs in the tuple $\overline{w}$ exactly $\#z$ times. In $(\star\star)$ we applied the decomposition lemma (Lemma 4.12). Every assignment β belongs to $\widetilde{\mathcal{E}}^{[\emptyset]}$ if and only if it can be decomposed into assignment $\beta = \beta_1 \cup \cdots \cup \beta_s$ such that there is an item $\iota_j \in \mathcal{L}^{[\emptyset]}_{y_j}$ with $\beta_j \in \widetilde{\mathcal{E}}^{\iota_j}$. In $(\star\star\star)$ we use the distributive law and the fact that $\beta_j \subseteq \beta$ and $\mathrm{dom}(\beta_j) \supseteq W_{y_j}$. In $(\star\star\star\star)$ we apply the fact from

Equation 7.2. Note that the condition for Equation 7.2 are given above. This concludes the correctness proof of $Q_{(w_1,\cdots,w_{\widetilde{d}})}(D)$.

We show now how to use these queries to obtain the result for Lemma 7.3. Let $S := \{\overline{w} \in \text{vars}(Q)^{\widetilde{d}},\ \widetilde{d} \leqslant 2d\}$. Note that $|S| \leqslant (|\text{vars}(Q)|+1)^{2d}$. We use in parallel for every query $Q_{\overline{w}}$ for every $\overline{w} \in S$ on the database D the data structure from Theorem 3.7. Note that $t_a = O(1)$ for each such query.

For the initialisation routine we initialise the data structure for every query $Q_{\overline{w}}$ for every $\overline{w} \in S$. This takes time $O((|\text{vars}(Q)|+1)^{2d}) = \mathit{poly}(Q)^{2d+1}$.

Every time the database receives an update, we update each data structure for $Q_{\overline{w}}$ for all $\overline{w} \in S$. This takes time $O((|\text{vars}(Q)|+1)^{2d}\ \mathit{poly}(Q)) = \mathit{poly}(Q)^{2d+1}$.

When we receive as input a tuple $\overline{w} \in S$ then we enumerate the tuple in time $O(1)$. Clearly,

$$Q_{(w_1,\cdots,w_{\widetilde{d}})}(D) \ = \ \sum_{\beta \in \widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{\widetilde{d}} \beta(w_p)^{\#w_p}.$$

It takes time $\mathit{poly}(Q)$ to output the value. This concludes the proof of Lemma 7.3. □

The proof of Theorem 3.8 follows from Lemma 7.3.

Proof of Theorem 3.8. Recall that for all $(s_1,\cdots,s_d) \in \mathcal{I}_d$ is

$$\mathsf{J}(\xi)_{1,(s_1,\cdots,s_d)} = \sum_{(j_1,\ldots,j_d)\in\mathcal{I}_d} \theta_{(j_1,\cdots,j_d)} \sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{d} \beta(x_{j_p}) \cdot \beta(x_{s_p}) - \sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \beta(y) \prod_{p=1}^{d} \beta(x_{j_p})$$

We use additionally the data structure of Lemma 7.3 for Q and D that can be initialised in time $t_i = O((|\text{vars}(Q)|+1)^{2d+1})$ and updated in time $t_u = O(\mathit{poly}(Q)^{2d+1})$. To receive the $\mathsf{J}(\xi)_{1,(s_1,\cdots,s_d)}$ very fast, we compute $\sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \beta(y) \prod_{p=1}^{d} \beta(x_{j_p})$ and $\sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{d} \beta(x_{j_p})\beta(x_{s_p})$. Let $\overline{w}^s$ ($\overline{w}^j$) be the tuple, we obtain, if we remove every 0 from $(s_1,\cdots,s_d)$ ($(j_1,\cdots,j_d)$). Clearly, $\sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \beta(y) \prod_{p=1}^{d} \beta(x_{j_p}) = Q_{\overline{w}^s}$ and $\sum_{\beta\in\widetilde{\mathcal{E}}^{[\emptyset]}} \prod_{p=1}^{d} \beta(x_{j_p})\beta(x_{s_p}) = Q_{\overline{w}^j}$. This can be done in time $O(1)$ and to compute the complete jacobi-matrix it takes time $|\mathcal{I}_d|^2 = \mathit{poly}(Q)^{2d+1}$. Thus, it takes time $\mathit{poly}(Q)^{2d+1}$ to do the forward pass. □

8. Outputting the jth solution for q-hierarchical conjunctive queries under updates

In this chapter, we show how the data structure for q-hierarchical conjunctive queries can be used to enhance for a given number $j \in \mathbb{N}$, the jth tuple in the enumeration with respect to a set of linear order $\{\leqslant^v\}_{v \in V}$. Here, we will achieve logarithmic (rather the constant) update time and answering time.

We will prove Theorem 3.9 that is restated in the following.

Theorem 3.9. *There is a dynamic algorithm that receives a q-hierarchical k-ary CQ with aggregates Q with aggregation time t_a and a σ-db D_0 of size $\|D_0\|$, and computes within preprocessing time poly$(Q) \cdot O(\|D_0\| \log(\|D_0\|))t_a$ a data structure that can be updated in time poly$(Q) \cdot O(\log(\|D\|))t_a$ and allows the following:*

(a) output the query result size $|Q(D)|$ in time $O(1)$ and

(b) enumerate the tuples in $Q(D)$ in lexicographical order with delay poly(Q) and

(c) test for an input tuple $\overline{a} \in \mathbf{dom}^{k+\ell}$ if $\overline{a} \in Q(D)$ within time $t_t =$ poly(Q) and

(d) upon input of a tuple $\overline{b} \in \mathbf{dom}^{k+\ell}$ output the tuple

$$\max\left\{\overline{a} \in Q(D) \ : \ \overline{a} \leqslant \overline{b}\right\}$$

if it exists, or a `SmallerThanMinimum`-message otherwise in time $t_l =$ poly(Q) and

(e) upon input of an arbitrary number j, take time poly$(Q) \cdot O(\log(\|D\|))$ to immediately output the jth tuple that the enumeration procedure of (b) would output and

(f) upon input of an arbitrary tuple $\overline{a} \in Q(D)$, take time poly$(Q) \cdot O(\log(\|D\|))$ to immediately output the number j such that $\overline{a}$ is the jth tuple the enumeration procedure of (b) *would output,*

where D is the current database.

In the next lemma, we consider the special case for the jth problem and for the jth-reverse problem and for queries without aggregates. Using this lemma in combination with Lemma 4.31, we will receive that result in Theorem 8.1 by applying the reduction lemma (Lemma 6.18).

Lemma 8.1. *There is a dynamic algorithm that receives a q-hierarchical k-ary CQ Q (without aggregates) and a σ-db D_0 of size $\|D_0\|$ and computes within preprocessing time $\text{poly}(Q) \cdot O(\|D_0\| \log(\|D_0\|))$ a data structure that can be updated in time $\text{poly}(Q) \cdot O(\log(\|D\|))$ and allows the following:*

(a) output the query result size $|Q(D)|$ in time $O(1)$,

(b) enumerate the tuples in $Q(D)$ in lexicographical order with delay $\text{poly}(Q)$,

(c) test for an input tuple $\overline{a} \in \mathbf{dom}^{k+\ell}$ if $\overline{a} \in Q(D)$ within time $t_t = \text{poly}(Q)$,

(d) upon input of an arbitrary number j, take time $\text{poly}(Q) \cdot O(\log(\|D\|))$ to immediately output the jth tuple that the enumeration procedure of (b) would output if $j \leqslant |Q(D)|$ and otherwise an `OutOfRange`-message,

(e) upon input of an arbitrary tuple $\overline{a} \in Q(D)$, take time $\text{poly}(Q) \cdot O(\log(\|D\|))$ to immediately output the number j such that $\overline{a}$ is the jth tuple the enumeration procedure of (b) *would output,*

where D is the current database.

A description of the data structure of Lemma 8.1 and a proof for part (d) is given in Section 8.1 and a proof for part (e) is given in Section 8.3.

In the Section 8.4 we give a proof for Theorem 3.9.

In the last section of this chapter (Section 8.5) we prove the lower bound stated in Theorem 3.16.

8.1. Output the jth solution

This section is devoted to the proof of Lemma 8.1(d).

We start with an example. We consider the q-hierarchical query

$$Q \;=\; \{(x, y_1, z_1, y_2, z_2, z_3, y_3) \;:\; R(x, y_1, z_1) \wedge S(x, y_2, z_2) \wedge T(x, y_2, z_3) \wedge U(x, y_3)\}.$$

Let D be the database with

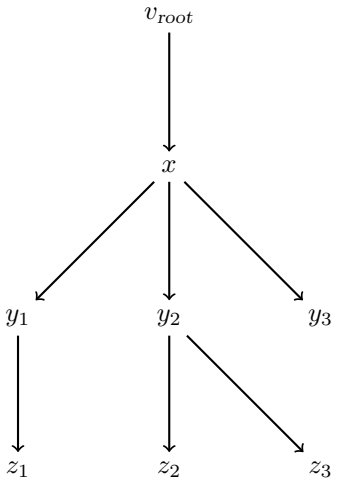
$$\begin{aligned} R^D &:= \{(1,4,1),\ (1,4,2),\ (1,5,1),\ (1,5,2),\ (2,4,9),\ (2,4,5),\\ &\qquad (3,1,4),\ (3,2,5),\ (3,3,6)\}\\ S^D &:= \{(1,1,3),\ (1,2,5),\ (2,1,2),\ (3,7,7),\ (3,7,8)\}\\ T^D &:= \{(1,1,4),\ (1,2,6),\ (2,1,3),\ (3,7,9),\ (3,7,10)\}\\ U^D &:= \{(1,7),\ (2,8),\ (2,9),\ (3,1)\} \end{aligned}$$

Table 8.1 shows the tuples that will be output by the enumeration algorithm. See Figure 8.1 for a q-tree of Q and Figure 8.2 for an illustration of the data structure generated for Q and D. In Figure 8.2, the red number above an item i is the number $\left|\widetilde{\mathcal{E}}^i\right|$ (in Section 6.5 it is described how to compute such values).

Table 8.1.: Output of the enumeration of $Q(D)$.

No.	x	y_1	z_1	y_2	z_2	z_3	y_3
1	1	4	1	1	3	4	7
2	1	4	1	2	5	6	7
3	1	4	2	1	3	4	7
4	1	4	2	2	5	6	7
5	1	5	1	1	3	4	7
6	1	5	1	2	5	6	7
7	1	5	2	1	3	4	7
8	1	5	2	2	5	6	7
9	2	4	9	1	2	3	8
10	2	4	9	1	2	3	9
11	2	4	5	1	2	3	8
12	2	4	5	1	2	3	9
13	3	1	4	7	7	9	1
14	3	1	4	7	7	10	1
15	3	1	4	7	8	9	1
16	3	1	4	7	8	10	1
17	3	2	5	7	7	9	1
18	3	2	5	7	7	10	1
19	3	2	5	7	8	9	1
20	3	2	5	7	8	10	1
21	3	3	6	7	7	9	1
22	3	3	6	7	7	10	1
23	3	3	6	7	8	9	1
24	3	3	6	7	8	10	1

Figure 8.1.: q-tree for the example query



In this running example, we are now interested in the question: What is the jth tuple that will be output by the enumeration algorithm? Upon input of a number j, we want to quickly provide an answer to the question. For the rest of this example, let us assume that $j = 20$.

The value $\left|\widetilde{\mathcal{E}}^i\right|$ of the first item in the x-list of the *start*-item is 8. Therefore, we know that the enumeration procedure starts with 8 valuations α where $\alpha(x) = 1$. Afterwards, the enumeration algorithm jumps to the next item in the start list to enumerate all the tuples where x is assigned to 2. As we know from its count value, there are 4 such tuples. Therefore, the component for x in the first 20 tuples in the enumeration are not assigned with 1 or 2. But in the enumeration there will follow 12 tuples where x is assigned to 3 and since $8 + 4 < 20 \leqslant 8 + 4 + 12$ (these values are taken from the $\left|\widetilde{\mathcal{E}}^i\right|$ values from the items in the x-list of $[\emptyset]$), we know that x must be assigned with 3. Note that the 20th tuple in the enumeration is exactly the $20 - 12 = 8$th tuple in the enumeration from $[\frac{3}{x}]$. So, we try to find the 8th tuple in the enumeration from $[\frac{3}{x}]$.

Now, we continue with the item $[\frac{3}{x}]$ and check in its lists how many assignments they will enumerate. We start with the y_3-list (on the right) and we see that there is only one assignment, namely $\beta(y_3) = 1$. Now, we know that for all 12 tuples which are enumerated from $[\frac{3}{x}]$, the variable y_3 will be assigned with 1. Then, we consider the y_2-list in the middle and we see that it has 4 assignments to enumerate. The assignments are $\beta_1 = \frac{7\ 7\ 9}{y_2 z_2 z_3}, \beta_2 = \frac{7\ 7\ 10}{y_2 z_2 z_3}, \beta_3 = \frac{7\ 8\ 9}{y_2 z_2 z_3}, \beta_4 = \frac{7\ 8\ 10}{y_2 z_2 z_3}$. Now, in the first tuple of the enumeration from $[\frac{3}{x}]$ the variables y_2, z_2, z_3 are assigned with the values of β_1, the second tuple with the values of β_2, the third tuple with the values of β_3, the fourth tuple with the values of β_4 and this repeats for three times. To find out

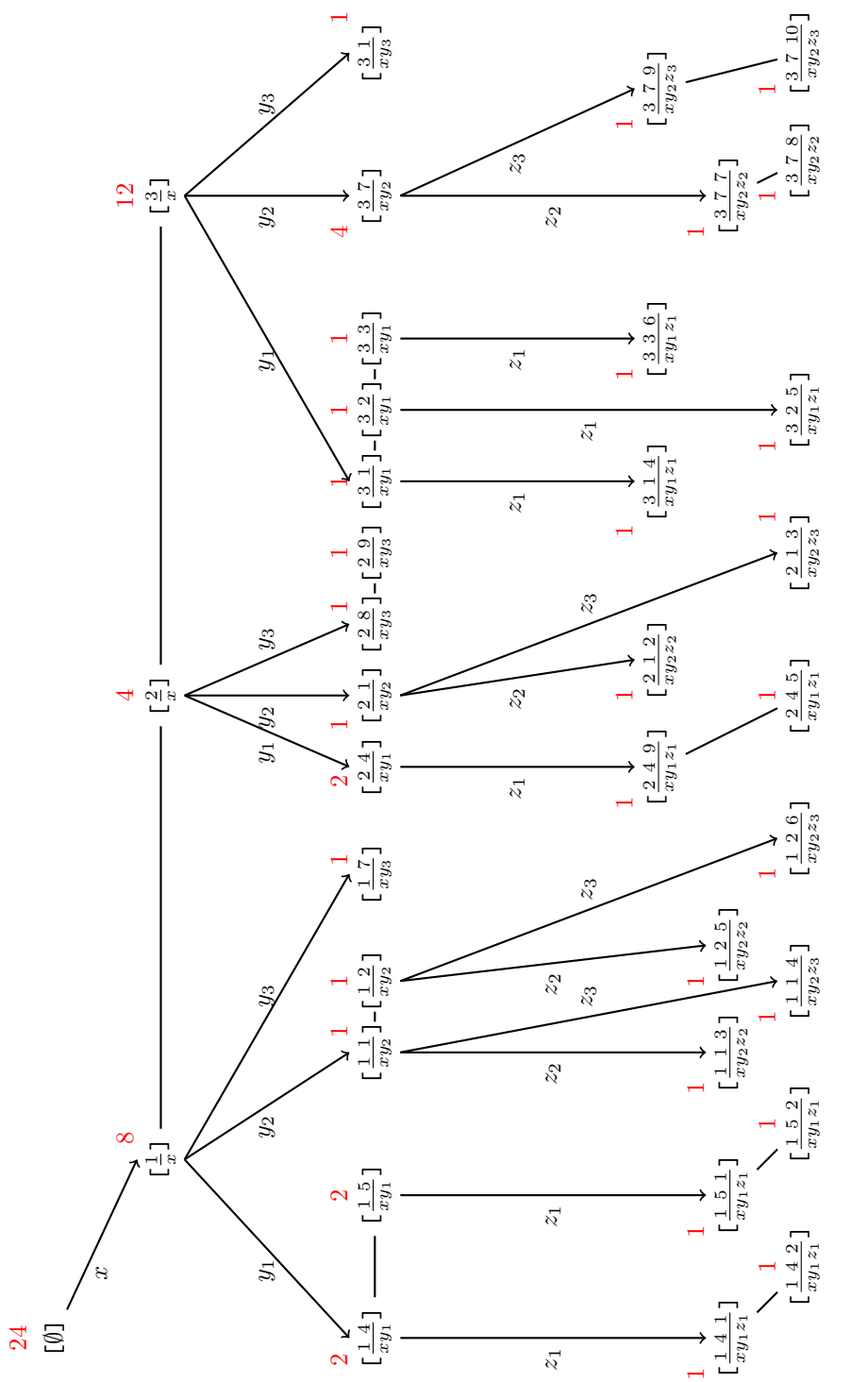


Figure 8.2.: Illustration of the data structure of the example database.

which is the right assignment for tuple number 8, we simply compute 8 modulo 4. We receive 0 and then we know that β_4 is the correct assignment, since for every 4th tuple in the enumeration the variables y_3, z_2, z_3 are assigned with the values of β_4. Finally, we have to take into account the y_1-list. It has the three assignments $\gamma_1 = \frac{1\ 4}{y_1 z_1}, \gamma_2 = \frac{2\ 5}{y_1 z_1}, \gamma_3 = \frac{3\ 6}{y_1 z_1}$. The enumeration will output for 4 times γ_1 (in combination with every β_j), then output for 4 times γ_2 and afterwards 4 times γ_3. It is straightforward to see that γ_2 is the assignment, we are looking for. To compute the right solution we divide 8 by 4 and we receive the information that we need to output γ_2. (Note that we have to round up the solution of the division). Thus, we obtain the solution tuple $(3, 2, 5, 7, 8, 10, 1)$.

By using the idea illustrated in this example, we can enrich the data structure in Theorem 3.3 to obtain a proof for Lemma 8.1.

For the remainder of this section we assume that $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is a q-hierarchical conjunctive query, $\text{vars}(Q) = \{x_1, \ldots, x_m\}$ with $0 \leqslant k \leqslant m$, and Q is of the form

$$Q \quad = \quad \left\{(x_1 \ldots x_k, b_1, \cdots, b_\ell) \ : \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right)\right\}, \tag{8.1}$$

where $b_1, \ldots, b_\ell \in \mathbf{dom}$ and $\psi_1, \ldots, \psi_d$ are atomic queries of schema σ.

We need that data structure from Section 6.6 for the counting query

$$Q_c := \{(\mathsf{prod}(c_{x_1}, \ldots, c_{x_m})) \ : \ \varphi\}$$

from Lemma 6.11 available.

We need the following variants of the modulo operation and the division operation. For all integers $a, b \in \mathbb{N}$ let

$$a \ \widehat{\mathbf{mod}} \ b := \begin{cases} b & \text{if } a \bmod b \equiv 0 \\ a \bmod b & \text{otherwise} \end{cases}$$

and

$$a \ \widehat{\mathbf{div}} \ b := \lceil a/b \rceil \ .$$

To compute for a given number $j \in \mathbb{N}$ the jth tuple of the enumeration of $Q(D)$, we start the procedure GetAssign of Algorithm 11 with j and the *start*-item as arguments.

To implement line 10 we use the following lemma:

Lemma 8.2. *There is a dynamic algorithm that receives as input a list $\langle n_1, \ldots, n_s \rangle$ of type $\mathbb{N}$ and computes within $t_p = O(s \cdot \log s)$ preprocessing time a data structure that can be updated in time $t_u = O(\log s)$ and allows on input of a number $m \in \mathbb{N}$ either*

- *output the index $p \in [s]$ such that $\sum_{i=1}^{p-1} n_i < m \leqslant \sum_{i=1}^{p} n_i$ and the number $\sum_{i=1}^{p-1} n_i$ if $m \leqslant \sum_{i=1}^{s} n_i$ or*
- *output the message `OutOfRange` if $m > \sum_{i=1}^{s} n_i$.*

Algorithm 11 Algorithm to output the j tuple.

1: **function** GETASSIGN($j, i.\{<^v\}_{v\in\mathsf{succ}(v^i)}$)
2: **Input:** Number $j \in [\left|\widetilde{\mathcal{E}}^i\right|]$, item i
3: **Output:** The assignment for the jth tuple of enumerating $\widetilde{\mathcal{E}}^i$.
4: **if** v^i is a leaf in the q-tree **then return** α^i
5: Let $u_1 <^{v^i} \cdots <^{v^i} u_d$ be the children of v^i in T_{free}.
6: **for all** $p \in [d]$ **do**
7: $c_p \leftarrow \prod_{q=p+1}^{d} \sum_{\iota\in\mathcal{L}^i_{u_q}} \left|\widetilde{\mathcal{E}}^\iota\right|$
8: $m_p = j \ \widehat{\mathbf{div}}\ c_p$
9: $n_p = m_p \ \widehat{\mathbf{mod}}\ \sum_{\iota\in\mathcal{L}^i_{u_q}} \left|\widetilde{\mathcal{E}}^\iota\right|$
10: For the list $\langle i_1, \cdots, i_s\rangle = \mathcal{L}^i_u$ compute the $q \in [d]$ such that $\sum_{r=1}^{q-1} \left|\widetilde{\mathcal{E}}^{i_r}\right| < n_p \leqslant \sum_{r=1}^{q} \left|\widetilde{\mathcal{E}}^{i_r}\right|$.
11: Let $\widehat{j}$ be $n_p - \sum_{r=1}^{q-1} \left|\widetilde{\mathcal{E}}^{i_r}\right|$.
12: $\alpha_p \leftarrow$ GETASSIGN$(\widehat{j}, i_q, \{<^v\}_{v\in\mathsf{succ}(u_p)})$
13: **return** $\alpha_1 \cup \cdots \cup \alpha_d$

and upon input of an item n_r from the list, output the number $\sum_{i=1}^{r} n_i$.

Note an update means here, an update on a list as described in page 13.

A proof for this lemma is given in Section 8.2.

To realise the algorithm, we need to get fast access to the values $\left|\widetilde{\mathcal{E}}^i\right|$ and to $\sum_{\iota\in\mathcal{L}^i_u} \left|\widetilde{\mathcal{E}}^i\right|$ for every child u of v^i in T_{free}. This can be done by using the data structure we obtain from the aggregate query that outputs the number of results in Section 6.5. Note that the data structure stores the numbers for each item on the item. Furthermore, we store for every item i and for every child u of v^i a list that contains the values $\langle\left|\widetilde{\mathcal{E}}^\iota\right| : \iota \in \mathcal{L}^i_u\rangle$ and the data structure from Lemma 8.2. The order of the elements in the list corresponds to the order of the items in the list, i.e., the mth element in the list is $|\left|\widetilde{\mathcal{E}}^\iota\right|\,|$ if and only if ι is the mth element in $\mathcal{L}^i_u$. The data structure can be used to get fast access to q in line 10 and to compute the value $\widehat{j}$ in line 11.

If we update the database we insert/delete a constant number of items and modify a constant number of $\left|\widetilde{\mathcal{E}}^i\right|$ values. For every item we modify in the data structure, we update the corresponding element in the data structure with Lemma 8.2. In particular, we need $poly(Q) \times O(\log(\|D\|))$ since we have to update $poly(Q)$ lists in $O(\log(\|D\|))$ time.

We show now that Algorithm 8.1 is correct.

Claim 8.3. *Upon input of an item i and $j \in [\left|\widetilde{\mathcal{E}}^i\right|]$ and a family of linear orderings*

$\{<^v\}_{v\in\mathsf{succ}(v^i)}$ *the function* GetAssign$(j, i, \{<^v\}_{v\in\mathsf{succ}(v^i)})$ *in Algorithm 11 outputs the* j*th assignment yielded by the function* Enum$(i, \{<^v\}_{v\in\mathsf{succ}(v^i)})$ *in Algorithm 3. Furthermore, Algorithm 11 takes time* $O(|\mathsf{succ}(v^i)|\,(|\mathsf{succ}(v^i)| + \log(|\mathrm{adom}(D)|)))$.

Proof. We show this claim by induction over the height of an item in T_{free}.

For the induction base, let us consider an item of height 0. Then, v^i is a leaf in T_{free} and $\widetilde{\mathcal{E}}^i = \{\alpha^i\}$ and $j = 1 = \left|\widetilde{\mathcal{E}}^i\right|$ and the function in GetAssign outputs α^i, that is the unique assignment yielded by Enum$(i, \emptyset)$. This takes time $O(1)$.

For the induction step let us consider an item of height h and let $u_1 <^{v^i} \cdots <^{v^i} u_d$ be the children of v^i in T_{free} and $<^{v^i}$ be a linear order we receive as input. Note by the decomposition lemma (Lemma 4.12) the jth assignment can be decomposed to $\alpha_1 \cup \cdots \cup \alpha_d$ where $\alpha_p \in \widetilde{\mathcal{E}}^{\iota_p}$ with $\iota_p \in \mathcal{L}^i_{u_p}$ for all $p \in [d]$. For all $p \in [d]$ it holds that α_p is taken as an element from the loops **for** $\iota_p \in \mathcal{L}^i_{u_p}$ **do for** $\alpha_p \in$ Enum$(\iota_p, \{<^v\}_{v\in\mathsf{succ}(v^{\iota_p})})$ **do**. By construction of the algorithm, the number of assignments β with $\beta \supseteq \alpha_p$ that will be yielded is c_p. Thus, the assignment we need is in the m_pth iteration. Note that there might be for-loops that force to restart the for-loops. But, the for-loops mentioned above have exactly $\sum_{\iota\in\mathcal{L}^i_{u_q}} \left|\widetilde{\mathcal{E}}^{\iota}\right|$ iterations. Thus, the assignment we need, is the n_pth assignment in the iteration of the for-loops mentioned above. In the next lines, the algorithm picks from the for-loops the item i_q that will be supposed in the n_pth iteration. Then, we subtract the number of iterations with the item that appears before i_q and take this number to receive the assignment α_p from GetAssign$(\widehat{j}, i_q, \{<^v\}_{v\in\mathsf{succ}(u_p)})$ that works by induction hypothesis correctly. Note that $1 \leqslant n_p \leqslant \left|\widetilde{\mathcal{E}}^{i_p}\right|$.

Line 7 takes time $O(d)$ and line 10 and 11 takes $O(\log(|\mathrm{adom}(D)|))$ time, since we use Lemma 8.2 to compute the values. By induction hypothesis line 12 takes time $O(|\mathsf{succ}(u_p)|\,(|\mathsf{succ}(u_p)| + \log(n)))$. The other lines in the body of the for-loop takes time $O(1)$. Since we use the data structure from Section 6.6 for the query Q_c we have constant access to the $\left|\widetilde{\mathcal{E}}^i\right|$ values. All in all, the time needed for the whole for-loop is

$$\sum_{p\in[d]} O(d) + O(\log(|\mathrm{adom}(D)|)) + O(|\mathsf{succ}(u_p)|\,(|\mathsf{succ}(u_p)| + \log(|\mathrm{adom}(D)|)))$$
$$= O(|\mathsf{succ}(v^i)|\,(|\mathsf{succ}(v^i)| + \log(|\mathrm{adom}(D)|)))$$

Therefore, it takes time $O(|\mathsf{succ}(v^i)|\,(|\mathsf{succ}(v^i)| + \log(|\mathrm{adom}(D)|)))$ to return the assignment. □

To conclude the proof of Lemma 8.1 we check upon input of a number $j \in \mathbb{N}$ if $j \leqslant |\varphi(D)|$ using the data structure for output the number $|\varphi(D)|$. This can be done in $O(1)$. If this is not the case, we output the `OutOfRange` message. Otherwise, we use Algorithm 11 with input j and the start item `[∅]` to obtain the jth assignment α of enumeration of $\varphi(D)$ in time $O(|\mathsf{succ}(v^i)|\,(|\mathsf{succ}(v^i)| + \log(n)))$ and output the tuple $(\alpha(x_1), \ldots, \alpha(x_k), b_1, \ldots, b_\ell)$. This concludes the proof of Lemma 8.1(d).

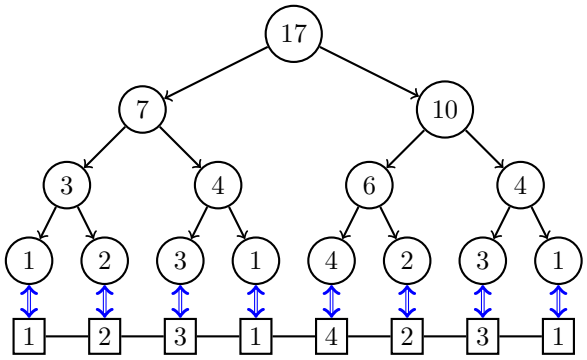


Figure 8.3.: Example of a sumtree.

8.2. Proof of Lemma 8.2

Let $\mathcal{L}$ be a list of integers of size s. The data structure we use is an enrichment of the data structure in Theorem 2.1, i.e., for the list $\mathcal{L}$ we have an balanced tree available, where every leaf is associated with an item in the list and the leafs are sorted as the elements in the list. Additionally, we store for every vertex v a field $\mathsf{num}(v)$ that is defined as follows: For all leafs v the number $\mathsf{num}(v)$ is the number of the item in the list it is indicated to. For all internal vertices v the number $\mathsf{num}(v)$ is the sum of the number values of its children. For all vertices v, we let $\mathcal{F}_v \subseteq [s]$ be the set of indices j such that there is a path from v to a leaf that indicates j. Note that it easily follows that $\mathsf{num}(v) = \sum_{j \in \mathcal{F}_v} \mathcal{L}[j]$. With $\mathcal{L}[j]$ we denote the jth element in the list $\mathcal{L}[j]$. See Figure 8.3 for the balanced tree for a list, where the numbers in the nodes v depict the number $\mathsf{num}(v)$.

We show on an example: upon input of the number 12 and the data structure in Figure 8.3, we show how to find the index p such that $\sum_{i=1}^{p-1} n_i < 12 \leqslant \sum_{i=1}^{p} n_i$. First of all we check on the root v whether its number is greater than 17. Since it is not the case, we now consider the root and check the number of its children. As we see, the left child v_{left} has the number 7. Since $7 = \sum_{j \in \mathcal{F}_v} \mathcal{L}[j]$, it follows that the sum of the first 4 elements in the list is 7. Thus, the index p, we search, must be greater than 4. In such a case we continue with the right vertex v_{right} and its left child w_{left} and its right child w_{right}. We also store the number 7 from v_{left} in a variable `current`. We consider now w_{left} and see that $\texttt{current} + \mathsf{num}(w_{\text{left}}) = 7 + 6 = 13 > 12$. In such a case we know that the sum of the first six elements in the list is greater than 12 and therefore for the p we search is $p \leqslant 6$. Note that $\mathsf{num}(w_{\text{left}})$ is the sum $\mathcal{L}[5] + \mathcal{L}[6]$. Then, we jump in the next step to the node w_{left} and consider its left child u_{left} and right child u_{right}. We do not change the `current` value. It still stores the sum of the values of the first four elements in the list. Again, we add `current` and $\mathsf{num}(u_{\text{left}})$ to

get the sum of the first five elements in the list 11. Since

$$\texttt{current} + \texttt{num}(u_{\text{left}}) = \sum_{i=1}^{5} \mathcal{L}[i] < 12 \leqslant \sum_{i=1}^{6\ (=p)} \mathcal{L}[i] = 13 = \texttt{current} + \texttt{num}(w_{\text{left}})$$

we have found the searched index $p = 6$.

Algorithm 8.2 shows how to solve this problem with the data structure. First of all, we check if the input number n is greater than the number value of the root. Note that the number on the root is $\sum_{i=1}^{s} \mathcal{L}[i]$. In that case, we output the `OutOfRange` message. Otherwise we start the procedure FIND in Algorithm 12 with input values 0 for `current`, m for j and the root node for v.

Algorithm 12 Procedure to find the index.

```
1:  function FIND(current, m, v)
2:      Input: current, m ∈ ℕ and vertex v
3:      if v is a leaf then
4:          return p is the index of element indicated by v and current.
5:      Let v_left be the left child of v
6:      Let v_right be the right child of v
7:      if m ⩽ current + num(v_left) then
8:          return FIND(current, m, v_left)
9:      else
10:         return FIND(current + num(v_left), m, v_right)
```

For all vertices v let $\mathit{first}(v) := \min \mathcal{F}_v$ and $\mathit{last}(v) := \max \mathcal{F}_v$. To show the correctness of the algorithm, we show by induction of the recursion depth of FIND that by calling the FIND($\texttt{current}, m, v$) procedure the following invariant holds:

- $$\sum_{i=1}^{\mathit{first}(v)-1} \mathcal{L}[i] < m \leqslant \sum_{i=1}^{\mathit{last}(v)} \mathcal{L}[i]$$
- $$\texttt{current} = \sum_{i=1}^{\mathit{first}(v)-1} \mathcal{L}[i]$$

Induction base $d = 1$. When we call FIND$(0, m, v_{root})$ at the beginning, we have made sure that $m \leqslant \sum_{i=1}^{s} \mathcal{L}[i]$. Note that since $\mathit{first}(v_{\text{root}}) = 1$ and $\mathit{last}(v_{\text{root}}) = s$, we have

$$\sum_{i=1}^{\mathit{first}(v_{root})-1} \mathcal{L}[i] = 0 < m \leqslant \sum_{i=1}^{s} \mathcal{L}[i] = \sum_{i=1}^{\mathit{last}(v_{\text{root}})} \mathcal{L}[i]$$

and

$$\texttt{current} = \sum_{i=1}^{\mathit{first}(v_{root})-1} \mathcal{L}[i] = 0\ .$$

Inductive step $d{-}1 \to d$. We call FIND($\texttt{current}, m, w$) during the execution of FIND($\texttt{current}', m, u$) that have called by the algorithm in recursion depth $d{-}1$. By induction hypothesis we have

$$\sum_{i=1}^{\mathit{first}(u)-1} \mathcal{L}[i] < m \leqslant \sum_{i=1}^{\mathit{last}(u)} \mathcal{L}[i]$$

and

$$\texttt{current}' = \sum_{i=1}^{\mathit{first}(u)-1} \mathcal{L}[i]$$

- **Case 1:** FIND($\texttt{current}, m, w$) was called by FIND($\texttt{current}', m, u$) since $m \leqslant \texttt{current}' + \texttt{num}(w)$ where w is the left child of u. Then $\texttt{current} = \texttt{current}'$. Since w is the left child of u, it follows that $\mathit{first}(w) = \mathit{first}(u)$ and therefore we have

$$\texttt{current} = \texttt{current}' = \sum_{i=1}^{\mathit{first}(u)-1} \mathcal{L}[i] = \sum_{i=1}^{\mathit{first}(w)-1} \mathcal{L}[i]$$

and

$$\begin{aligned}\sum_{i=1}^{\mathit{first}(w)-1} \mathcal{L}[i] &= \sum_{i=1}^{\mathit{first}(u)-1} \mathcal{L}[i] < m \leqslant \texttt{current}' + \texttt{num}(w) \\ &= \sum_{i=1}^{\mathit{first}(w)-1} \mathcal{L}[i] + \sum_{i=\mathit{first}(w)}^{\mathit{last}(w)} \mathcal{L}[i] = \sum_{i=1}^{\mathit{last}(w)} \mathcal{L}[i]\end{aligned}$$

- **Case 2:** FIND($\texttt{current}, m, w$) was called by FIND($\texttt{current}', m, u$) since $m > \texttt{current}' + \texttt{num}(v)$ where v is the left child of u. Then $\texttt{current} = \texttt{current}' + \texttt{num}(v)$ and w is the right child of u. Since w is the right child of u, it follows that $\mathit{last}(w) = \mathit{last}(u)$. Therefore, we have

$$m \leqslant \sum_{i=1}^{\mathit{last}(u)} \mathcal{L}[i] = \sum_{i=1}^{\mathit{last}(w)} \mathcal{L}[i]$$

By

$$\begin{aligned}m > \texttt{current}' + \texttt{num}(v) &= \sum_{i=1}^{\mathit{first}(u)-1} \mathcal{L}[i] + \sum_{i=\mathit{first}(v)}^{\mathit{last}(v)} \mathcal{L}[i] \\ \overset{\mathit{first}(v)=\mathit{first}(u)}{=} \sum_{i=1}^{\mathit{last}(v)} \mathcal{L}[i] &= \sum_{i=1}^{\mathit{first}(w)-1} \mathcal{L}[i]\end{aligned}$$

and

$$\texttt{current} = \texttt{current}' + \texttt{num}(v) = \sum_{i=1}^{\mathit{first}(w)-1} \mathcal{L}[i]$$

it follows that the invariant holds before FIND($\texttt{current}, m, w$) is called.

This concludes the induction.

Let v be a leaf and let p be the index of the element indicated by v. Then, we have $\mathit{first}(v) = \mathit{last}(v) = p$. In particular, we have

$$\sum_{i=1}^{\mathit{first}(v)-1} \mathcal{L}[i] = \sum_{i=1}^{p-1} \mathcal{L}[i] < m \leqslant \sum_{i=1}^{\mathit{last}(v)} \mathcal{L}[i] = m = \sum_{i=1}^{p} \mathcal{L}[i]$$

and

$$\texttt{current} = \sum_{i=1}^{\mathit{first}(v)-1} \mathcal{L}[i] = \sum_{i=1}^{p-1} \mathcal{L}[i]$$

This is exactly the index and `current` is the value we are searching for and this concludes the proof of the correctness of Algorithm 8.2.

To implement the routine that upon input of an item n_r from the list, outputs the number $\sum_{i=1}^{r} n_i$, we use Algorithm 13 on input of the leaf w in the sum tree that is associated to n_r and $\texttt{sum} = \mathsf{num}(w)$.

Algorithm 13 Procedure to compute the sum $\sum_{i=1}^{r} n_i$.

```
1:  function GetSum(current, m, w)
2:      Input: current ∈ ℕ and vertex w.
3:      if w is the root of the sum tree then
4:          return current
5:      else
6:          Let u be the parent of w.
7:          if w is the left child of u or the only child of u then
8:              GetSum(current, w).
9:          else
10:             Let u_left be the left child of u.
11:             GetSum(current + num(u_left), v).
```

For all vertices v let $\mathit{first}(v) := \min \mathcal{F}_v$ and $\mathit{last}(v) := \max \mathcal{F}_v$. To show the correctness of the algorithm, we prove the following. When we start GETSUM($\mathsf{num}(v), v$) where v is a leaf in the sum tree, then in every recursive step the following invariant holds. If GETSUM($\texttt{current}, u$) will be executed, we have

$$\texttt{current} = \sum_{i=\mathit{first}(u)}^{\mathit{last}(v)} \mathcal{L}[i].$$

We show this by an induction over the recursive depth. For the induction base let us consider the case that we start with GETSUM($\mathsf{num}(v), v$). Then the claim

holds since v is a leaf and thus

$$\texttt{current} = \mathsf{num}(v) = \sum_{i=\mathit{first}(v)}^{\mathit{last}(v)} \mathcal{L}[i].$$

For the inductive step, let us consider a recursive step with depth $d+1$.

- *Case 1:* GetSum($\texttt{current}, w$) was executed in GetSum($\texttt{current}, u$) (with recursive depth d) in line 8. Then w is the left child of u or the only child of u and thus $\mathit{first}(u) = \mathit{first}(w)$. We obtain from the induction hypothesis the following.

$$\texttt{current} = \sum_{i=\mathit{first}(u)}^{\mathit{last}(v)} \mathcal{L}[i] = \sum_{i=\mathit{first}(w)}^{\mathit{last}(v)} \mathcal{L}[i].$$

- *Case 2:* GetSum($\texttt{current}, w$) was executed in GetSum($\texttt{current}, u$) (with recursive depth d) in line 11. Then w is the right child of u and thus $\mathit{first}(w) = \mathit{last}(u_{\text{left}}) + 1$ where u_{left} is the left child of u. We obtain from the induction hypothesis and the definition of $\mathsf{num}(u_{\text{left}})$ the following.

$$\begin{aligned}
\texttt{current} + \mathsf{num}(u_{\text{left}}) &= \sum_{i=\mathit{first}(w)}^{\mathit{last}(v)} \mathcal{L}[i] + \sum_{i=\mathit{first}(u_{\text{left}})}^{\mathit{last}(u_{\text{left}})} \mathcal{L}[i] \\
&= \sum_{i=\mathit{last}(u_{\text{left}})+1}^{\mathit{last}(v)} \mathcal{L}[i] + \sum_{i=\mathit{first}(u_{\text{left}})}^{\mathit{last}(u_{\text{left}})} \mathcal{L}[i] \\
&= \sum_{i=\mathit{first}(u_{\text{left}})}^{\mathit{last}(v)} \mathcal{L}[i].
\end{aligned}$$

Thus, it follows that the number returned by GetSum($\mathsf{num}(v), v$) where v is the leaf for n_r is equal to $\sum_{i=1}^{r} n_i$. Since the recursion depth is bounded by the depth of the tree which is $O(\log(s))$, Algorithm 13 takes time $O(\log(s))$.

We show now how to maintain sum trees under updates.

Recall, that the data structure is an enrichment of the data structure of Theorem 2.1. In particular, the tree with the sums is a AVL-tree and during an update operation, one has to rotate the tree to get a balanced tree (see Figures 2.1 and 2.2). We enrich the implementation of the update steps as follows.

- When we insert or delete a node to/from the list, we update the $\mathsf{num}(v)$ value to all nodes on the path from the root to the corresponding left. This takes $O(s)$ time.
- After a closer look to the rotations operations (see [65]), we see that an rotation changes the children of at most three nodes. We change the algorithm for updating such that we recompute the number $\mathsf{num}(v)$ values. The operations for such an update takes time $O(1)$.

Therefore updating the data structure can be done in $O(\log(s))$.

This concludes the proof of Lemma 8.2.

8.3. jth reverse

In this section, we consider the jth reverse problem, i.e., we prove Lemma 8.1(e). For the remainder of this section we assume that $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ is a q-hierarchical conjunctive query, $\text{vars}(Q) = \{x_1, \ldots, x_m\}$ with $0 \leqslant k \leqslant m$, and Q is of the form

$$Q \quad = \quad \left\{(x_1 \ldots x_k, b_1, \cdots, b_\ell) \ : \ \exists x_{k+1} \cdots \exists x_m \left(\psi_1 \wedge \cdots \wedge \psi_d\right)\right\}, \tag{8.2}$$

where $b_1, \ldots, b_\ell \in \mathbf{dom}$ and $\psi_1, \ldots, \psi_d$ are atomic queries of schema σ. Let Q be a q-hierarchical query and let D be a database. We receive as input an assignment α and we request the number j such that $(\alpha(x_1), \ldots, \alpha(x_k), b_1, \ldots, b_\ell)$ is the jth tuple in the enumeration of $Q(D)$.

To compute the number, we define the number $\mathsf{jth}(v)$ for all $v \in \text{vars}(Q)$ such that $\mathsf{jth}(v)$ is the number of steps until we output the assignment $\alpha|_{\text{vpath}[v]} \cup \alpha|_{\mathsf{succ}(v)}$ in the procedure $\mathsf{Enum}(\texttt{[}\alpha|_{\text{vpath}[v]}\texttt{]}, \{<^w\}_{w \in \mathsf{succ}(v^{\texttt{[}\alpha|_{\text{vpath}[v]}\texttt{]}})})$. The number of steps means here number of assignments that will be yielded. The number $\mathsf{jth}(v)$ can be defined inductively over the height of v.

- If v has height 0, we have $\mathsf{jth}(v) = 0$.

- If v has height $h > 0$: Let $u_1, \ldots, u_s$ be the children of v in T_{free} and for all $j \in [s]$ let $\mathcal{I}_j$ be the set of items that appear before $\texttt{[}\alpha|_{\text{vpath}[u_j]}\texttt{]}$ in the u_j-list of $\texttt{[}\alpha|_{\text{vpath}[v]}\texttt{]}$. We count the number of steps in the procedure $\mathsf{Enum}(\texttt{[}\alpha|_{\text{vpath}[v]}\texttt{]}, \{<^w\}_{w \in \mathsf{succ}(v^{\texttt{[}\alpha|_{\text{vpath}[v]}\texttt{]}})})$ until we receive $\alpha_i = \alpha|_{\mathsf{succ}(u_i)}$ for all $i \in [s]$. By construction of the algorithm, we run the first two for-loops until $\alpha_1 = \alpha|_{\mathsf{succ}(u_1)}$, then the next two for-loops until $\alpha_2 = \alpha|_{\mathsf{succ}(u_2)}$ and so on, i.e., the number can be expressed as

$$\mathsf{jth}(v) \ = \ \sum_{j=1}^{s} \text{“number of steps until } \alpha_j = \alpha|_{\mathsf{succ}(u_j)} \text{ ”}.$$

To compute the number of steps until $\alpha_j = \alpha|_{\mathsf{succ}(u_j)}$ for all $j \in [s]$, we count the number of steps until we reach one step before $\alpha_j = \alpha|_{\mathsf{succ}(u_j)}$ for all $j \in [s]$. We add then 1 to obtain the number of steps until $\alpha_j = \alpha|_{\mathsf{succ}(u_j)}$. We need $\sum_{\iota \in \mathcal{I}_j} \left|\widetilde{\mathcal{E}}^\iota\right|$ steps until the for loop **for** $\hat{\iota}_j \in \mathcal{L}^i_{u_j}$ **do** reaches $\hat{\iota}_j = \texttt{[}\alpha|_{\text{vpath}[u_j]}\texttt{]}$. Then, by induction hypothesis, we need $\mathsf{jth}(u_j)$ steps until the for loop **for** $\alpha_j \in \textsc{Enum}(\hat{\iota}_j, \{<^v\}_{v \in \mathsf{succ}(u_j)})$ **do** reaches $\alpha_j = \alpha|_{\mathsf{succ}(u_j)}$. For every iteration we have

$\prod_{\widehat{j}=j+1}^{s} \sum_{\iota \in \mathcal{L}^{u_j}_{[\alpha|_{\mathrm{vpath}[v]}]}} \left|\widetilde{\mathcal{E}}^{\iota}\right|$ steps that run over the other for loops. We obtain

$$\mathsf{jth}(v) \;=\; \sum_{j=1}^{s} \left(\left(\sum_{\iota \in \mathcal{I}_j} \left|\widetilde{\mathcal{E}}^{\iota}\right| + \mathsf{jth}(u_j) - 1 \right) \cdot \prod_{\widehat{j}=j+1}^{s} \sum_{\iota \in \mathcal{L}^{[\alpha|_{\mathrm{vpath}[v]}]}_{u_{\widehat{j}}}} \left|\widetilde{\mathcal{E}}^{\iota}\right| + 1 \right)$$

To compute the numbers $\mathsf{jth}(v)$ we us a bottom up algorithm on the q-tree of Q. The numbers $\sum_{\iota \in \mathcal{I}_j} \left|\widetilde{\mathcal{E}}^{\iota}\right|$ and $\sum_{\iota \in \mathcal{L}^{[\alpha|_{\mathrm{vpath}[v]}]}_{u_j}} \left|\widetilde{\mathcal{E}}^{\iota}\right|$ can be computed from the sum tree described the first section of this chapter. Note that we have for every item i and for every $u \in \mathit{child}(v^i)$ a list with a sum tree available that contains for every $\iota \in \mathcal{L}^i_u$ the number $\left|\widetilde{\mathcal{E}}^{\iota}\right|$. From Lemma 8.2 it follows that there is a routine where we can output the sum $\sum_{\iota \in \mathcal{I}_j} \left|\widetilde{\mathcal{E}}^{\iota}\right|$ in time $O(\log(\mathrm{adom}(D)))$.

All in all, we need $O(\mathit{poly}(Q)\log(\mathrm{adom}(D)))$ time to compute the $\mathsf{jth}(v)$ values for all $v \in \mathrm{vars}(Q)$. This concludes the poof of Lemma 8.1(e).

8.4. Proof for Theorem 3.9

In the following corollary we combine data structure of Lemma 4.31 with the data structure of Lemma 8.1. Recall that the data structure of Lemma 4.31 is simply a lexicographically ordered data structure. The logarithmic factor in the update time comes from the fact, that one has to insert the items to the designated position since every list in the data structure is sorted. The number of items that we consider during an update step is equal to the number of items we would consider if the data structure was not lexicographically ordered. The logarithmic factor in the update step of Lemma 4.31 comes from the fact that we need logarithmic time to insert or delete an element to/from the corresponding sum tree. This has to be done for every item whose fit status changes during an update step. If the data structure is lexicographically ordered this has also to be done for the same number of items as in the non-lexicographically ordered structure. Thus, it follows, when considering a lexicographically ordered version of the data structure of Lemma 8.1, we obtain the following result.

Corollary 8.4. *There is a dynamic algorithm that receives a q-hierarchical k-ary CQ (without aggregates) $Q(x_1, \ldots, x_k, b_1, \ldots, b_\ell)$ and a σ-db D_0 of size $\|D_0\|$, and computes within preprocessing time $\mathit{poly}(Q) \cdot O(\|D_0\| \log(\|D_0\|))$ a data structure that can be updated in time $\mathit{poly}(Q) \cdot O(\log(\|D\|))$ and allows the following:*

(a) output the query result size $|Q(D)|$ in time $O(1)$,

(b) enumerate the tuples in $Q(D)$ in lexicographical order with delay $\mathit{poly}(Q)$,

(c) test for an input tuple $\overline{a} \in \mathbf{dom}^{k+\ell}$ if $\overline{a} \in Q(D)$ within time $t_t = \mathit{poly}(Q)$, and

8. Outputting the jth solution for q-hierarchical conjunctive queries under updates

(d) upon input of a tuple $\overline{b} \in \mathbf{dom}^{k+\ell}$ *output the tuple*

$$\max\left\{\overline{a} \in Q(D) \ : \ \overline{a} \leqslant \overline{b}\right\}$$

if it exists, or a `SmallerThanMinimum`*-message otherwise in time* $t_l = poly(Q)$*,*

(e) upon input of an arbitrary number j*, take time* $poly(Q) \cdot O(\log(\|D\|))$ *to immediately output the* j*th tuple that the enumeration procedure of (b) would output.*

(f) upon input of an arbitrary tuple $\overline{a} \in Q(D)$*, take time* $poly(Q) \cdot O(\log(\|D\|))$ *to immediately output the number* j *such that* $\overline{a}$ *is the* j*th tuple the enumeration procedure of* (b) *would output,*

where D *is the current database.*

Theorem 3.9 is the result of applying Lemma 6.18 on Corollary 8.4:

Suppose that we receive as input a q-hierarchical CQ with aggregates Q. To initialise the data structure for Q on database D the algorithm we obtain from Lemma 6.18(b) to compute a corresponding schema $\widetilde{\sigma}$ and a q-hierarchical query $\widetilde{Q}$ (without aggregates). Using Corollary 8.4, we obtain that there is a data structure for $\widetilde{Q}$ and $\widetilde{\sigma}$-dbs that can be initialised for the empty database in time $\widetilde{t_i} = poly(Q)$ and can be updated in time $\widetilde{t_u} = poly(\widetilde{Q}) \cdot O(\log(\|\widetilde{D}\|)) \leqslant poly(Q) \cdot O(\log(\|D\|))$. Using Lemma 6.18(c) we obtain an algorithm for Q with initialisation time $t_i = poly(Q)$ and update time $t_u = poly(Q) \cdot O(\log(\|D\|))t_a$. Note that the data structure from Corollary 8.4 for $\widetilde{Q}$ on $\widetilde{D}$ is present.

The **count**-routine can be done as follows. Use the **count**-routine from Corollary 8.4(a) for $\widetilde{Q}$ and $\widetilde{D}$. Since $Q(D) = \widetilde{Q}(\widetilde{D})$ (Lemma 6.18(a)), it follows that we output the number $\widetilde{Q}(\widetilde{D})$.

The **enumerate**-routine can be done as follows. Use the **enumerate**-routine from Corollary 8.4(b) for $\widetilde{Q}$ and $\widetilde{D}$. Since $Q(D) = \widetilde{Q}(\widetilde{D})$ (Lemma 6.18(a)), it follows that we enumerate the tuples in $Q(D)$. Moreover, the tuples will be enumerated lexicographically.

The **test**-routine can be done as follows. Use the **test**-routine from Corollary 8.4(b) for $\widetilde{Q}$ and $\widetilde{D}$. Since $Q(D) = \widetilde{Q}(\widetilde{D})$ (Lemma 6.18(a)), it follows that we test for the tuples in $Q(D)$ upon input of an tuple.

The routine in Theorem 3.9(d) for computing the next smaller tuple can be done as follows. Use the corresponding routine form from Corollary 8.4(e). Since $Q(D) = \widetilde{Q}(\widetilde{D})$ (Lemma 6.18(a)), it follows that we obtain the next smaller one from $Q(D)$.

The **jth**-routine can be done as follows. Use the **jth**-routine from Corollary 8.4(e). Since we use the **enumerate** routine from Corollary 8.4(b) it follows that we output the jth tuple the enumeration routine would output.

The **jth**-reverse-routine can be done as follows. Use the **jth**-reverse-routine from Corollary 8.4(f). Since we use the **enumerate** routine from Corollary 8.4(b) it follows that we output the jth tuple the enumeration routine would output.

8.5. Lower bounds

This section is devoted to the proof of Theorem 3.16.

For proving the theorem the following lemma will be convenient:

Lemma 8.5. *Let Q be a CQ and let D be a database and $\leqslant$ be a linear order on the result set $Q(D)$. Suppose that there is an Algorithm $\mathtt{A}_{\leqslant}$ that outputs in time t_j on input of a number $j \in \mathbb{N}$ the tuple $\overline{a} \in Q(D)$ such that $\overline{a}$ is the jth tuple with respect to $\leqslant$ if $j \leqslant |Q(D)|$ or outputs the `OutOfRange`-Message otherwise. Then, there is an algorithm `B` that enumerates $Q(D)$ with delay $t_d = t_j$.*

Proof. See Algorithm 14 for Algorithm `B`.

Algorithm 14 Algorithm `B`.

```
1: j = 1
2: Let ā be the output of A_≤ on input j.
3: while ā is not the OutOfRange-Message do
4:     Output ā.
5:     j ← j + 1
6:     Let ā be the output of A_< on input j.
7: Output end-of-enumeration message EOE.
```

Clearly, `B` enumerates $Q(D)$ with delay t_j. □

Proof of Theorem 3.16. Suppose for a contradiction that there is an algorithm that receives a self join free conjunctive query Q that is not q-hierarchical and a database D, and computes with arbitrary preprocessing time a data structure that can be updated in time $O(n^{1-\varepsilon})$ and outputs in time $O(n^{1-\varepsilon})$ on input of a $j \in \mathbb{N}$ the tuple $\overline{a} \in Q(D)$ such that $\overline{a}$ is the jth tuple with respect to a linear order $\leqslant$ on $Q(D)$ if $j \leqslant |Q(D)|$ or outputs the `OutOfRange`-Message otherwise for any linear order of the result set $Q(D)$. Applying Lemma 8.5, we can enumerate the tuples in $Q(D)$ with delay $O(n^{1-\varepsilon})$. This violates to Theorem 3.13. □

This concludes the proof of Theorem 3.16.

9. Answering unions of conjunctive queries under updates

In this chapter we consider dynamic query evaluation for UCQs. The syntax and semantics of UCQs are given in page 20. To transfer our notions of *hierarchical* queries from CQs to UCQs, we say that a UCQ $Q(\overline{u})$ of the form $Q_1(\overline{u}_1) \cup \cdots \cup Q_d(\overline{u}_d)$ is q-hierarchical (t-hierarchical) if every CQ $Q_i(\overline{u}_i)$ in the union is q-hierarchical (t-hierarchical). Note that for *Boolean* queries (CQs as well as UCQs) the notions of being q-hierarchical and being t-hierarchical coincide, and for a k-ary UCQ Q it can be checked in time $poly(Q)$ if Q is q-hierarchical or t-hierarchical.

In the next section we consider the task of testing and enumerating UCQs. In Section 9.2 we identify the UCQs for which the task of reporting the jth query result can be done efficiently under updates.

9.1. Enumerating and testing UCQs

Testing. This paragraph is based on [21]. Theorem 3.10 generalises the statement of Theorem 3.6 from CQs to UCQs. Its proof follows easily from the Theorems 3.2 and 3.6. Theorem 3.10 is restated in the following.

Theorem 3.10 ([21]). *There is a dynamic algorithm that receives a t-hierarchical k-ary UCQ Q and a σ-db D_0, and computes within $t_p = poly(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows to test for an input tuple $\overline{a} \in \mathbf{dom}^k k$ if $\overline{a} \in Q(D)$ within time $t_t = poly(Q)$. Furthermore, the algorithm allows to answer a t-hierarchical Boolean UCQ within time $t_{ans} = O(1)$.*

Proof of Theorem 3.10. It follows immediately from Theorem 3.6 (and Theorem 3.2 for the statement on *Boolean* UCQs), as we can maintain all CQs in the union in parallel and then decide whether at least one of them is satisfied by the current database and the given tuple. □

Enumerating. It turns out that q-hierarchical UCQs, like q-hierarchical CQs, allow for efficient enumeration under updates. This is stated in Theorem 3.11 restated in the following.

Theorem 3.11 ([21]). *There is a dynamic algorithm that receives a q-hierarchical k-ary UCQ Q and a σ-db D_0, and computes within $t_p = poly(Q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(Q)$ and allows to enumerate $Q(D)$ with delay $t_d = poly(Q)$.*

In contrast to Theorem 3.10, the result does not follow immediately from the tractability of the enumeration problem for q-hierarchical CQs, because one has to ensure that tuples from result sets of two different CQs are not reported twice while enumerating their union.

In the following an alternative proof of Theorem 3.11 as in [21] is given.

To prove Theorem 3.11, we first show a general method for enumerating the union of sets which is stated in Lemma 9.1. This idea relies on the algorithm for enumerating the union of two sets in [39].

Lemma 9.1. *Let $S_1, \ldots, S_m$ be m sets. Suppose that there is for every $i \in [m]$ an algorithm $\mathtt{A}^i_{enum}$ that enumerates the elements in S_i with delay t_d, and an algorithm $\mathtt{A}^i_{\mathsf{test}}$ that can test in time t_t for a given element a whether $a \in S_i$. Then, there is an algorithm that enumerates the elements in $\bigcup_{i\in[m]} S_i$ without repetition with delay $O(n(t_t + t_d))$.*

To illustrate the idea of Lemma 9.1 we consider the following example.

Example 9.2. *Let us consider the three sets $S_1 := \{1, 4, 5\}$, $S_2 := \{2, 4\}$, $S_3 := \{2, 3, 4, 5\}$ and we have for each $i \in [3]$ an algorithm $\mathtt{A}^i_{\mathsf{enum}}$ that enumerates the set S_i with delay t_d and an algorithm $\mathtt{A}^i_{\mathsf{test}}$ which upon input of a natural number n tests if $n \in S_i$. The aim is to enumerate $S_1 \cup S_2 \cup S_3$ without repetition. The algorithm for the general case is given in Algorithm 15. First of all, we set for all $i \in [3]$ the element s_i to the first element of S_i, i.e. $s_1 = 1, s_2 = 2, s_3 = 2$. Then, we start with the set S_1 (with $\ell = 1$ in the for-loop in line 2) and start with* OUTPUT(1). *In the procedure we check in the for-loop if there is a $K \in \{2, 3\}$ with $1 \in S_K$, i.e., if $1 \in S_2 \cup S_3$. Since this is not the case, we print 1 and set s_1 to 4 and since $s_1 \neq$ EOE we again execute* OUTPUT(1). *Since $4 \in S_2$ the if-condition in line 9 holds for $K = 2$. Thus, we continue with setting s_1 to 5 and start with* OUTPUT(2). *Since $s_2 = 2 \in S_3$, the if condition in line 9 holds for $K = 3$ and set s_2 to 4 and continue with* OUTPUT(3). *This procedure prints $s_3 = 2$ and sets s_3 to 3. Then, we continue with the while-loop since $s_1 \neq$ EOE and since $5 \in S_3$ we set s_1 to EOE and execute* OUTPUT(3) *which prints 3 and sets s_3 to 4. Since $s_1 =$ EOE, the for-loop in line 2 sets ℓ to 2 and thus, we consider the set S_2. Since $4 \in S_3$, we print $s_3 = 4$ and set s_3 to the next element in S_3, which is 5 and s_2 gets the value EOE and afterwards we enumerate 5 from S_3 and then the algorithm terminates since all elements in S_3 were printed and $s_3 =$ EOE.*

The elements that were printed in OUTPUT(1) *belong to $S_1 \setminus (S_2 \cup S_3)$, the elements that were printed in* OUTPUT(2) *belong to $S_2 \setminus S_3$ and the elements in* OUTPUT(3)*, that were printed, belong to S_3. Altogether, we enumerated the set $S_1 \cup S_2 \cup S_3$.*

We consider now the general case.

Proof of Lemma 9.1. Algorithm 15 shows how to enumerate the set $S_1 \cup S_2 \cup \cdots \cup S_m$.

Correctness Every time we call OUTPUT(ℓ), we print the element s_ℓ if and only if there is no $K \in \{\ell + 1, \cdots, m\}$ such that $s_\ell \in S_K$, i.e., $s_\ell \notin \bigcup_{K=\ell+1}^{m} S_K$. The fact

Algorithm 15 Algorithm for enumerating $S_1 \cup S_2 \cup \cdots \cup S_m$.

```
1: For all i ∈ [m] let s_i be the first element of S_i.
2: for ℓ = 1 to m do
3:     while s_ℓ ≠ EOE do
4:         OUTPUT(ℓ)
5: print EOE
6:
7: procedure OUTPUT(ℓ)
8:     for K = ℓ + 1 to m do
9:         if s_ℓ ∈ S_K then
10:            Set s_ℓ to the successor element of s_ℓ in S_ℓ
11:            OUTPUT(K)
12:            return
13:    print s_ℓ
14:    Set s_ℓ to the successor element in S_ℓ
```

that we initialise s_ℓ to the first element and then we set s_ℓ to its successor element in every call of OUTPUT(ℓ) and that the for and while loop at the beginning ensures that we repeat the call OUTPUT(ℓ) until $s_\ell = \texttt{EOE}$ for all $\ell \in [m]$, guarantees that we consider every element in S_ℓ exactly once. All in all, the elements that are printed in the whole algorithm are

$$S_1 \setminus \left(\bigcup_{K=2}^{m} S_K \right) \cup \cdots \cup S_m = \bigcup_{\ell=1}^{m} \left[S_\ell \setminus \left(\bigcup_{K=\ell+1}^{m} S_K \right) \right] = \bigcup_{\ell=1}^{m} S_\ell$$

To show that we enumerate the elements without repetition, let us assume for a contradiction that there are $\ell_1, \ell_2 \in [m]$ with $\ell_1 < \ell_2$ and an element $s \in S_{\ell_1} \cap S_{\ell_2}$ such that s will be printed during a call of OUTPUT(ℓ_1) and a call of OUTPUT(ℓ_2). Since $s \in S_{\ell_2}$ and $\ell_2 > \ell_1$ this will not be print by a call of OUTPUT(ℓ_1). Furthermore, since we enumerate the sets themselves without repetition, we enumerate the disjunction without repetition.

Running time First of all, we show that every time we call OUTPUT(ℓ) it holds that $s_\ell \neq \texttt{EOE}$. If OUTPUT(ℓ) will be called in line 4 this is guaranteed by the condition of the while loop, and if it is called in line 11 there is a $k < \ell$ and an element $s \in S_k$ such that $m \in S_\ell$ and $s \notin S_{k+1} \cup \ldots \cup S_{\ell-1}$ (since otherwise the algorithm calls OUTPUT(p) [instead of OUTPUT(ℓ)] for the smallest $p \in \{k+1, \ldots, \ell-1\}$ with $m \in S_p$). The number of times that the procedure OUTPUT(ℓ) will be called in line 11 is at most $|(S_1 \cup \cdots \cup S_{\ell-1}) \cap S_\ell| \leqslant |S_\ell|$ (since there must be an element $s_\ell \in S_\ell$ and a $k \in [\ell-1]$ such that $s_\ell \in S_k \setminus (S_{k+1} \cup \cdots \cup S_{\ell-1})$. In particular, every element in S_ℓ will be mentioned in $S_1 \cup \cdots \cup S_{\ell-1}$ once). Therefore $s_\ell \neq \texttt{EOE}$ if it is called in line 11. When we call OUTPUT(ℓ), we either output an element or there is a $K > \ell$ such that OUTPUT(K) will be called. Since $K \leqslant m$, we need at most m steps until we output

an element, after a call of OUTPUT(ℓ). During the call, we have at most $K - \ell$ tests in time $O(t_t)$ and we jump to the next element in a set in time $O(t_d)$. Therefore, it takes time $O(m(t_d + t_t))$ after a call of the OUTPUT procedure until we output an element. By construction of the algorithm, we have $O(m(t_d + t_t))$ delay. □

Using Lemma 9.1 we obtain a proof for Theorem 3.11:

Proof of Theorem 3.11. Let $Q = Q_1 \cup \cdots \cup Q_m$ be the input query. Note that $\|Q_i\| \leqslant \|Q\|$. We store in parallel for every $i \in [m]$ the data structure from Theorem 3.3 for the q-hierarchical query Q_i on D_0. Every time the database receives an update we update the data structure for every $i \in [m]$ using the update procedure from Theorem 3.3. This takes time $O(m \cdot \mathit{poly}(Q)) = O(\mathit{poly}(Q))$. To enumerate the result sets $Q_i(D)$ use the enumeration algorithm from Theorem 3.3(b) to enumerate the tuples with delay $t_d = O(\mathit{poly}(Q))$. Furthermore, with Theorem 3.3(a) we can test time $t_t = O(\mathit{poly}(Q))$, whether a tuple belongs to the result set $Q_i(D)$. Applying the enumeration and testing routines to Lemma 9.1 we obtain an enumeration algorithm for $Q(D)$ that enumerates the tuples without repetition and with delay $O(m(\mathit{poly}(Q) + \mathit{poly}(Q))) = O(\mathit{poly}(Q))$.
This concludes the proof of Theorem 3.11. □

9.2. Reporting the jth solution

In this section we identify a class of UCQs for which we can maintain the jth solution problem under updates. Here we consider strongly exhaustively q-hierarchial queries.

Recall that a union of conjunctive queries $\bigcup_{i\in[m]} Q_i$ is strongly exhaustively q-hierarchial if there is a q-hierarchical query Q' with $Q'(D) = \bigcap_{i\in[m]} Q_i(D)$ for all σ-dbs D.

Let $Q = Q_1 \cup \cdots \cup Q_m$ be an arbitrary strongly exhaustively q-hierarchial union of conjunctive queries.

In the following Theorem 3.12 is restated.

Theorem 3.12. *There is a dynamic algorithm that receives a strongly exhaustively q-hierarchial k-ary UCQ Q and a σ-db D_0 of size $\|D_0\|$, and computes within preprocessing time* $\exp(Q) \cdot O(\|D_0\| \log(\|D_0\|))$ *a data structure that can be updated in time* $\exp(Q) \cdot O(\log(\|D\|))$ *and allows the following:*

(a) enumerate the tuples in $Q(D)$ with delay $\mathit{poly}(Q)$ and

(b) upon input of an arbitrary number j, take time $\exp(Q) \cdot O(\log(\|D\|))$ *to immediately output the jth tuple that the enumeration procedure of (a) would output,*

where D is the current database.

To prove Theorem 3.12 we use the following lemma that enables to output the jth solution when using the algorithm from Lemma 9.1.

Lemma 9.3. *Let X be a set and let $S_1, \ldots, S_m$ be m subsets of X and for all $\emptyset \neq \mathcal{I} \subseteq [m]$ let $S_{\mathcal{I}} := \bigcap_{i\in\mathcal{I}} S_i$. Assume that for all $\emptyset \neq \mathcal{I} \subseteq [m]$ we have the following algorithms available.*

(a) An algorithm $\mathtt{A}^{\mathcal{I}}_{enum}$ *that enumerates elements in* $S_{\mathcal{I}}$ *with delay* t_d *in lexicographical order and*

(b) an algorithm $\mathtt{A}^{\mathcal{I}}_{count}$ *that outputs the number* $|S_{\mathcal{I}}|$ *in time* t_c *and*

(c) an algorithm $\mathtt{A}^{\mathcal{I}}_{test}$ *that upon input of an element* $x \in X$ *tests in time* t_t *if* $x \in S_{\mathcal{I}}$ *and*

(d) an algorithm $\mathtt{A}^{\mathcal{I}}_{jth}$ *that upon input of a number* $j \in \mathbb{N}$ *returns the* j*th element in* $S_{\mathcal{I}}$ *that the algorithm* $\mathtt{A}^{\mathcal{I}}_{enum}$ *will output in time* t_j *if it exists or a* `OutOfRange`*-message otherwise and*

(e) an algorithm $\mathtt{A}^{\mathcal{I}}_{jth\text{-}rev}$ *that upon input of an element* $a \in S_{\mathcal{I}}$ *outputs the number* j *in time* t_{jr} *such that* a *is the* j*th element algorithm* $\mathtt{A}^{\mathcal{I}}_{enum}$ *will output and*

(f) an algorithm $\mathtt{A}^{\mathcal{I}}_{lex}$ *that upon input of an element* $x \in X$ *outputs in time* t_l *the element* $\max\{a : a \in S_{\mathcal{I}},\ a \leqslant x\}$ *if it exists or a* `SmallerThanMinimum` *message otherwise.*

Then, there is an algorithm that upon input of a number $j \in \mathbb{N}$ *outputs the* j*th element that the algorithm in Lemma 9.1 for enumerating* $S_1 \cup \cdots \cup S_m$*, by applying the enumeration algorithms* $\mathtt{A}^{\{i\}}_{enum}$ *and testing algorithms* $\mathtt{A}^{\{i\}}_{test}$ *for all* $i \in [m]$*, will output if* $j \leqslant |S_1 \cup \cdots \cup S_m|$ *or output an* `OutOfRange`*-message otherwise. The algorithm takes time* $O\left(m^2 2^m (t_j + t_l + t_{jr} + t_c) + n(t_t + t_j)\right)$.

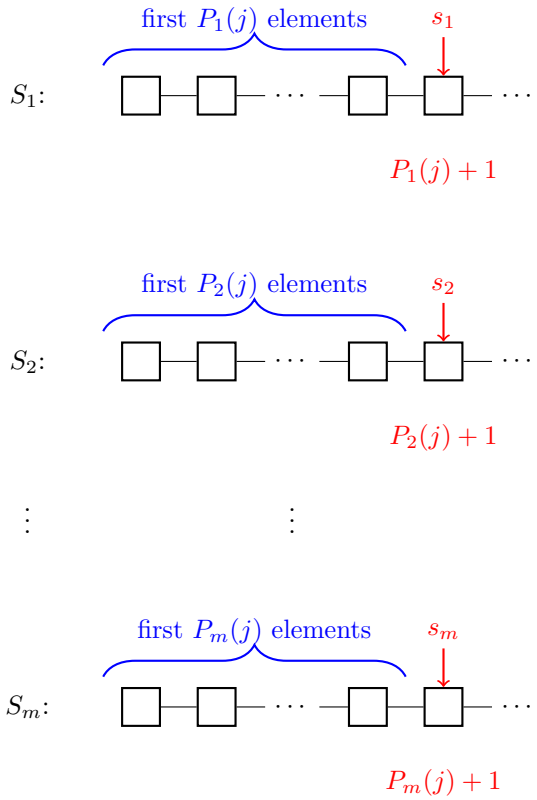
The jth step is the fragment of Algorithm 15 that enumerates the jth element. To be precise, the jth step starts, when we start the OUTPUT(ℓ) procedure in line 4 in Algorithm 15 that ends by printing the jth element.

To prove Lemma 9.3, we try to simulate the jth step of Algorithm 15, i.e., the fragment of the algorithm that enumerates the jth element. To do this, we need to compute the current elements s_i for all $i \in [m]$ and the current number $\ell \in [m]$ in Algorithm 15 before the jth step starts.

To compute these elements we define for all $k \in [m]$ and $j \in \mathbb{N}$ numbers $P_k(j)$ such that the following holds. The elements s_k (this is the current element of the set S_k in Algorithm 15) is set to the $(P_k(j)+1)$th element in S_k for all $k \in [m]$ when the jth step starts. In particular, the number of elements in S_k we have already considered, i.e., printed or skipped is $P_k(j)$. Note that the element s_k, i.e., the $(P_k(j)+1)$th element of S_k, was neither printed nor skipped. See Figure 9.1 for an illustration of the numbers $P_k(j)$ when the jth step starts. We will later describe how to compute the number $P_k(j)$. To determine the number ℓ in the for-loop in line 2 in Algorithm 15 we use the number j in the following way. For all $i \in [j]$ one has to check if $|S_1 \cup \cdots \cup S_{i-1}| < j \leqslant |S_1 \cup \cdots \cup S_k|$. The number i for which this condition holds true is the sought number for ℓ. The set S_ℓ is the current set which is considered since already $|S_1 \cup \cdots \cup S_{\ell-1}|$ elements were printed by the algorithm and, in particular, all elements in $S_1 \cup \cdots \cup S_{\ell-1}$ were either printed or skipped by the algorithm, but not all elements in S_ℓ.

The algorithm that simulates the jth step of Algorithm 15 is given in Algorithm 16. We will later describe how to implement the lines of Algorithm 15.

Figure 9.1.: Illustration of how the numbers $P_k(j)$ are set at the beginning of the jth step (for the special case where $\ell = 1$).



Algorithm 16 Algorithm to output the jth element of $S_1 \cup \cdots \cup S_m$.

Input: $j \in \mathbb{N}$.
if $j > |S_1 \cup \cdots \cup S_m|$ **then**
    Output `OutOfRange`-message
Let ℓ be the number such that $|S_1 \cup \cdots \cup S_{\ell-1}| < j \leqslant |S_1 \cup \cdots \cup S_\ell|$.
Let a be the $(P_\ell(j)+1)$th element of S_ℓ.
Let $k := \ell$.
while there is a $K > k$ such that $a \in S_K$ **do**
    $k := \min\{K : K > k \text{ and } a \in S_K\}$
    Let a be the $(P_k(j)+1)$th element of S_k.
Output a

In the jth step, we iterate over the elements in S_ℓ and therefore we consider the $(P_\ell(j)+1)$th element of S_ℓ. (Note that by definition of ℓ we have $P_\ell(j)+1 \leqslant |S_1 \cup \cdots \cup S_\ell|$.) This is the ℓ of the first for-loop in Algorithm 15 when enumerating the jth element. Then, the enumeration algorithm checks (in the OUTPUT procedure in Algorithm 15) for $k \in \{\ell+1, \cdots, m\}$ in ascending order if $a \in S_k$ holds. If we find such a k, we jump to the next element in S_k that was not output or skipped by the enumeration algorithm. This is exactly the $(P_k(j)+1)$th element of S_k. The enumeration algorithm repeats this until we cannot find an element that appears in a set with a higher index. This is exactly the element that is output by Algorithm 16 upon input of the number j.

The hard part is to compute the numbers $P_k(j)$. We will use the following notation. For all $k \in [m]$ let $a_{P_k(j)}$ be the $P_k(j)$th element in the enumeration of S_k. Furthermore, let

$$\widetilde{S}_k \;:=\; \begin{cases} \{s \in S_k \;:\; s \leqslant a_{P_k(j)}\} & \text{if } k \geqslant \ell \\ S_k & \text{otherwise} \end{cases}.$$

In other words: $\widetilde{S}_k$ is the set of the elements in S_k that Algorithm 15 already printed or skipped until it started the jth step. Before we go into detail for the general case let us consider the following example.

Example 9.4. *Recall Example 9.2. Our aim is now to output the* 3*rd element. Algorithm 16 sets* $\ell = 1$ *and since we iterate over the elements in* S_1*, we have to consider the* 3*rd element in* S_1*, i.e.,* $P_1(j) = j = 3$*. We consider* 5 *which is the* 3*rd element in* S_1 *and since* $5 \in S_3$ *we jump to the set* S_3*. Now we count how often we jumped to* Output(3) *during the enumeration of the first two elements. There are two cases why this could happen. The first case is that there is an element* a *in* $\widetilde{S}_1$ *(one of the elements in* S_1 *we already considered) which appears in* S_3 *but not in* S_2*. Note that if* a *appeared in* S_2 *we would first jump to* OUTPUT(2) *and not to* S_3*. This is exactly the case when considering* $5 \in S_1$ *in Example 9.2. The case can be written as follows. There is an element* $a \in (\widetilde{S}_1 \setminus S_2) \cap S_3$*. The second case is that there is an element in* S_2*, which we already considered, that also appears in* S_3*, i.e., there is an element in the set* $\widetilde{S}_2 \cap S_3$*. All in all, we can express the number* $P_3(j)$ *as*

$$\begin{aligned} P_3(j) &= \left|((\widetilde{S}_1 \setminus S_2) \cap S_3) \;\cup\; (\widetilde{S}_2 \cap S_3)\right| \\ &\overset{(\star)}{=} \left|(\widetilde{S}_1 \setminus S_2) \cap S_3\right| + \left|\widetilde{S}_2 \cap S_3\right| \\ &\overset{(\star\star)}{=} \left|(\widetilde{S}_1 \cap S_3) \setminus (\widetilde{S}_1 \cap S_2 \cap S_3)\right| + \left|\widetilde{S}_2 \cap S_3\right| \\ &\overset{(\star\star\star)}{=} \left|\widetilde{S}_1 \cap S_3\right| - \left|\widetilde{S}_1 \cap S_2 \cap S_3\right| + \left|\widetilde{S}_2 \cap S_3\right| \end{aligned}$$

Equation $(\star)$ *follows from the fact that the unions are disjoint, equation* $(\star\star)$ *by the de Morgan's law and* $(\star\star\star)$ *from the fact that* $(\widetilde{S}_1 \cap S_2 \cap S_3)$ *is a subset of* $(\widetilde{S}_1 \cap S_3)$*.*

It remains to show how to compute these cardinalities. In the following we show how to compute the number $\left|(\widetilde{S}_1 \cap S_2 \cap S_3)\right|$. *Suppose that we know the number* $P_1(j)$. *Then, we use algorithm* $\mathtt{A}_{jth}^{\{1\}}$ *to compute the* $P_1(j)$*th element* a *of* S_1. *Then, using algorithm* $\mathtt{A}_{lex}^{\{1,2,3\}}$ *we compute the maximum element* $b \in (\widetilde{S}_1 \cap S_2 \cap S_3)$ *which is lexicographically smaller than or equal to* a. *If such an element does not exist,* $\left|(\widetilde{S}_1 \cap S_2 \cap S_3)\right| = 0$. *Otherwise, we obtain the number* $\left|(\widetilde{S}_1 \cap S_2 \cap S_3)\right|$ *by applying* $\mathtt{A}_{jth\text{-}rev}^{\{1,2,3\}}$ *upon input of* b. *This is the number of elements in* $S_1 \cap S_2 \cap S_3$ *that are smaller than the current* s_1 *value (the* $P_1(j)$*th element in* S_1*) . The number* $\left|\widetilde{S}_2 \cap S_3\right|$ *can be computed by a similar way and the cardinality of* $S_2 \cap S_3$ *can simply be determined by using the* $\mathtt{A}_{count}^{\{2,3\}}$ *algorithm.*

We consider now the general case. To compute the number $P_k(j)$ it is necessary to have the $P_i(j)$ value available for all $i < k$. Hence, we compute the number $P_i(j)$ in ascending order, i.e., for $i = 1, 2, \ldots, k$. In the following it is described how to obtain a formula to compute the number $P_j(k)$ for $k > \ell$. We have to count the number of elements in S_k we have jumped to, when considering the previous elements. An element in S_k will only be considered if there is an $u \leqslant k$ such that there is an element $a \in S_u$ which also appears in S_k but not in $S_{u+1} \cup \cdots \cup S_{k-1}$ (since otherwise the OUTPUT operation would jump to the corresponding set S_q with $q \in \{u+1, \ldots, k-1\}$). In other words, there is an $i < k$ such that there is an element $a \in (S_u \setminus \bigcup_{j=u+1}^{k-1} S_j) \cap S_k$. To describe the number $P_k(j)$ one has to count the number of element for which the conditions above holds. Therefore, we obtain for $k > \ell$ the following

$$P_k(j) \;=\; \left| \bigcup_{u=1}^{k-1} \left(\widetilde{S}_u \setminus \bigcup_{i=u+1}^{k-1} S_i \right) \cap S_k \right| . \tag{9.1}$$

By using some transformations of the formula above, we obtain the following for-

mula.

$$P_k(j) = \left|\bigcup_{u=1}^{k-1}\left(\widetilde{S}_u \setminus \bigcup_{i=u+1}^{k-1} S_i\right) \cap S_k\right| \tag{9.2}$$

$$= \left|\bigcup_{u=1}^{k-1}\left((\widetilde{S}_u \cap S_k) \setminus \left(\bigcup_{i=u+1}^{k-1} S_i \cap \widetilde{S}_u \cap S_k\right)\right)\right| \tag{9.3}$$

$$\overset{(\star)}{=} \sum_{u=1}^{k-1}\left|(\widetilde{S}_u \cap S_k) \setminus \left(\bigcup_{i=u+1}^{k-1} S_i \cap \widetilde{S}_u \cap S_k\right)\right| \tag{9.4}$$

$$\overset{(\star\star)}{=} \sum_{u=1}^{k-1}\left(\left|\widetilde{S}_u \cap S_k\right| - \left|\bigcup_{i=u+1}^{k-1} S_i \cap \widetilde{S}_u \cap S_k\right|\right) \tag{9.5}$$

$$\overset{(\star\star\star)}{=} \sum_{u=1}^{k-1}\left(\left|\widetilde{S}_u \cap S_k\right| - \sum_{\emptyset\neq\mathcal{I}\subseteq\{u+1,\ldots,k-1\}} (-1)^{|\mathcal{I}|-1}\left|\bigcap_{i\in\mathcal{I}} S_i \cap \widetilde{S}_u \cap S_k\right|\right) \tag{9.6}$$

$$= \sum_{u=1}^{k-1}\left(\left|\widetilde{S}_u \cap S_k\right| + \sum_{\emptyset\neq\mathcal{I}\subseteq[k-1]\setminus[u]} (-1)^{|\mathcal{I}|}\left|\bigcap_{i\in\mathcal{I}} S_i \cap \widetilde{S}_u \cap S_k\right|\right) \tag{9.7}$$

The equation $(\star)$ follows from the fact that the unions are disjoint and $(\star\star)$ from the fact that set $\bigcup_{i=u+1}^{k-1} S_i \cap \widetilde{S}_u \cap S_k$ is a subset of $(\widetilde{S}_u \cap S_k)$. In $(\star\star\star)$ we used the inclusion-exclusion principle (see Equation 2.2).

Claim 9.5. *Let ℓ be the number such that $|S_1 \cup \cdots \cup S_{\ell-1}| < j \leqslant |S_1 \cup \cdots \cup S_\ell|$. Then, the following holds.*

- $$P_\ell(j) \;=\; j - \sum_{\emptyset\neq\mathcal{I}\subseteq[\ell]} (-1)^{|I|+1}\left|\bigcap_{i\in\mathcal{I}} S_i\right| \tag{9.8}$$

  and

- *for all $k > \ell$ we have*

$$P_k(j) \;=\; \sum_{u=1}^{k-1}\left(\left|\widetilde{S}_u \cap S_k\right| + \sum_{\emptyset\neq\mathcal{I}\subseteq[k-1]\setminus[u]} (-1)^{|\mathcal{I}|}\left|\bigcap_{i\in\mathcal{I}} S_i \cap \widetilde{S}_u \cap S_k\right|\right). \tag{9.9}$$

Proof. The case for $P_k(j)$ with $k \in [m] \setminus [\ell]$ follows immediately from Equation (9.7).

S_ℓ is the set which is iterated during the jth step. The number $j - |S_1 \cup \cdots \cup S_{\ell-1}|$ is the number of elements we have to consider in S_ℓ until we reach the jth element in $S_1 \cup \cdots \cup S_\ell$, after the elements in $|S_1 \cup \cdots \cup S_{\ell-1}|$ were enumerated. The number of elements in S_ℓ we considered until we enumerated the elements in $S_1 \cup \cdots \cup S_{\ell-1}$ is $\left|\left(\bigcup_{u=1}^{\ell-1} S_u \setminus \bigcup_{i=u+1}^{\ell-1} S_i\right) \cap S_\ell\right|$.

In summary, the number $P_\ell(j)$ can be determined as follows.

$$\begin{aligned}
P_\ell(j) \;&=\; j - |S_1 \cup \cdots \cup S_{\ell-1}| + \left|\left(\bigcup_{u=1}^{\ell-1} S_u \setminus \bigcup_{i=u+1}^{\ell-1} S_i\right) \cap S_\ell\right| \\
&=\; j - \sum_{\emptyset \neq \mathcal{I} \subseteq [\ell-1]} (-1)^{|I|+1} \left|\bigcap_{i\in\mathcal{I}} S_i\right| \;+\; \left|\left(\bigcup_{u=1}^{\ell-1} [S_u \cap S_\ell] \setminus \bigcup_{i=u+1}^{\ell-1} [S_i \cap S_\ell \cap S_u]\right)\right| \\
&\overset{(\star)}{=}\; j - \sum_{\emptyset \neq \mathcal{I} \subseteq [\ell-1]} (-1)^{|I|+1} \left|\bigcap_{i\in\mathcal{I}} S_i\right| \;+\; \sum_{u=1}^{\ell-1} \left|\left([S_u \cap S_\ell] \setminus \bigcup_{i=u+1}^{\ell-1} [S_i \cap S_\ell \cap S_u]\right)\right| \\
&\overset{(\star\star)}{=}\; j - \sum_{\emptyset \neq \mathcal{I} \subseteq [\ell-1]} (-1)^{|I|+1} \left|\bigcap_{i\in\mathcal{I}} S_i\right| \;+\; \sum_{u=1}^{\ell-1} |S_u \cap S_\ell| - \left|\bigcup_{i=u+1}^{\ell-1} [S_i \cap S_\ell \cap S_u]\right| \\
&\overset{(\star\star\star)}{=}\; j - \sum_{\emptyset \neq \mathcal{I} \subseteq [\ell-1]} (-1)^{|I|+1} \left|\bigcap_{i\in\mathcal{I}} S_i\right| \;+\; \sum_{u=1}^{\ell-1} |S_u \cap S_\ell| \\
&\qquad - \sum_{\emptyset \neq \mathcal{I} \subseteq [\ell-1]} (-1)^{|\mathcal{I}+1|} \left|\bigcap_{i\in\mathcal{I}} S_i \cap S_\ell \cap S_u\right|
\end{aligned}$$

The equation $(\star)$ follows from the fact that the unions are disjoint and $(\star\star)$ from the fact that set $\bigcup_{i=u+1}^{k-1} S_i \cap S_\ell \cap S_u$ is a subset of $(S_\ell \cap S_u)$. In $(\star\star\star)$ we used the inclusion-exclusion principle (see Equation 2.2). □

To avoid notational clutter, we define for every non-empty set $\mathcal{I} \subseteq [m]$ the following set.

$$S_\mathcal{I} \;:=\; \bigcap_{i\in\mathcal{I}} S_i$$

We obtain the numbers $|S_\mathcal{I}|$ from $\mathtt{A}^{\mathcal{I}}_{\mathsf{count}}$ in time t_c. Since we compute the numbers $|S_\mathcal{I}|$ for all $\emptyset \neq \mathcal{I} \subseteq [\ell]$ it takes time $2^\ell t_c$ to compute $P_\ell(j)$.

To compute $P_k(j)$ for all $k > \ell$ we use the formula from Claim 9.5. To compute the cardinality of sets of the form $\left|\bigcap_{i\in\mathcal{I}} S_i \;\cap\; \widetilde{S}_u\right|$ for $u \in [m]$ and $\mathcal{I} \subseteq [m]$ we do the following. If S_i is of the form $\widetilde{S}_i = S_i$, we simply use algorithm $\mathtt{A}^{\mathcal{I}\cup\{u\}}_{\mathsf{count}}$. In the other case, we use Algorithm 17.

Let $a_1, \ldots, a_{P_u(j)}$ be the first $P_u(j)$ elements of S_u enumerated by $\mathtt{A}^{\{u\}}_{\mathsf{enum}}$. In particular, we have $a_1 < \ldots < a_{P_u(j)}$. Note that

$$\bigcap_{i\in\mathcal{I}} S_i \;\cap\; \widetilde{S}_u \;=\; \left\{c \in \left\{a_1, \ldots, a_{P_u(j)}\right\} \;:\; c \in S_\mathcal{I}\right\}$$

Clearly, by definition of $S_\mathcal{I} \cap \widetilde{S}_u$ all elements in the set are smaller than or equal to $a_{P_u(j)}$. Then, for b in line 2 of Algorithm 17 we have

$$b = \max\left\{c \in \left\{a_1, \ldots, a_{P_u(j)}\right\} \;:\; c \in S_\mathcal{I}\right\}$$

Algorithm 17 Algorithm for computing $\left|\bigcap_{i\in\mathcal{I}} S_i \cap \widetilde{S}_u\right|$.

1: Use $\mathtt{A}_{\mathsf{jth}}^{\{u\}}$ to get the $P_u(j)$th element of S_u. Let $a_{P_u(j)}$ be the element.
2: Use $\mathtt{A}_{\mathsf{lex}}^{\mathcal{I}\cup\{u\}}$ to get b, the maximum element in $S_{\mathcal{I}\cup\{u\}}$ which is smaller than (or equal to) $a_{P_u(j)}$.
3: **if** $b ==$ `SmallerThanMinimum` **then**
4: **return** 0
5: Use $\mathtt{A}_{\mathsf{jth-rev}}^{\mathcal{I}\cup\{u\}}$ to get the number $\widehat{j}$, such that b is the $\widehat{j}$th element in the enumeration of $S_{\mathcal{I}\cup\{u\}}$
6: **return** $\widehat{j}$

If such an element does not exists, $\left|S_i \cap \widetilde{S}_u\right| = 0$. Since $\mathtt{A}_{\mathsf{enum}}^{\mathcal{I}\cup\{u\}}$ enumerates the elements in lexicographical order, the number we obtain in $\mathtt{A}_{\mathsf{jth-rev}}^{\mathcal{I}\cup\{u\}}$ is equal to the cardinality of $S_{\mathcal{I}} \cap \widetilde{S}_u$ (since we enumerate all elements in $c \in \left\{a_1, \ldots, a_{P_u(j)}\right\}$ with $c \in S_{\mathcal{I}}$ until b will be enumerated).

Algorithm 17 takes time $O(t_j + t_l + t_{jr})$ to compute cardinalities of sets of the form $\bigcap_{i\in\mathcal{I}} S_i \cap \widetilde{S}_u$. The number of sets of such a form in the formula of $P_u(j)$ (see Equation 9.8 and 9.9) is at most $(k-1)2^{k-1}$. All in all, the algorithm takes time $O((k-1)2^{k-1}(t_j + t_l + t_{jr} + t_c)$.

To execute Algorithm 16, we have to compute $P_k(j)$ for all $k \in \{\ell, \ldots, n\}$. Therefore, it takes time at most $O(m(m-1)2^{m-1}(t_j + t_l + t_{jr} + t_c))$ to compute all these values.

In summary, Algorithm 16 takes time

$$O(\underbrace{m(m-1)2^{m-1}(t_j + t_l + t_{jr} + t_c)}_{\text{compute } P_k(j) \text{ for all } k\in[m]\setminus[\ell]} + \underbrace{m2^m t_c}_{\text{line 4}} + \underbrace{t_j}_{\text{line 5}} + \underbrace{m(t_t + t_j)}_{\text{while-loop}})$$
$$= O(m^2 2^m (t_j + t_l + t_{jr} + t_c) + m(t_t + t_j))$$

This concludes the proof of Lemma 9.3 .

In the remainder of this section we will prove Theorem 3.12 where we apply Theorem 3.9 on Lemma 9.1 and 9.3.

To use Theorem 3.9 on Lemma 9.1, we will show that for a strongly exhaustively q-hierarchial query $\bigcup_{i=1}^{m} Q_i$ it holds that for every $\emptyset \neq \mathcal{I} \subseteq [m]$ the query $Q_{\mathcal{I}}$ that expresses that

$$Q_{\mathcal{I}}(D) = \bigcap_{i\in\mathcal{I}} Q_i(D) \text{ for all } \sigma\text{-dbs } D$$

is also q-hierarchical. This fact follows from the following claim

Claim 9.6. *Let the query*

$$Q := \{(x_1, \ldots, x_k, b_1, \ldots, b_\ell) : \exists y_1 \cdots \exists y_\ell R_1\overline{z}_1 \wedge \cdots \wedge R_m\overline{z}_m\}$$

be a q-hierarchical query. Then, the query

$$Q' := \left\{(x_{i_1}, \ldots, x_{i_{k'}}, b_{u_1}, \ldots, b_{u_{\ell'}}) : \exists y_{j_1} \cdots \exists y_{j_{\ell'}} R_1\overline{z}_1 \wedge \cdots \wedge R_{m-1}\overline{z}_{m-1}\right\}$$

where

- $\{x_{i_1}, \ldots, x_{i_{k'}}\} = \{x_1, \ldots, x_k\} \cap \bigcup_{M \in [m-1]} \{z : z \text{ is in } \overline{z}_M\}$ *and*
- $\{y_{j_1}, \ldots, y_{j_{\ell'}}\} = \{y_1, \ldots, y_\ell\} \cap \bigcup_{M \in [m-1]} \{z : z \text{ is in } \overline{z}_M\}$ *and*
- $\{b_{u_1}, \ldots, b_{u_{\ell'}}\} = \{b_1, \ldots, b_\ell\} \cap \bigcup_{M \in [m-1]} \{z : z \text{ is in } \overline{z}_M\}$ *and*

let T be a q-tree of Q and T' be the induced subtree of T on $\mathrm{vars}(Q')$*. Then, T' is a q-tree for Q'. In particular, Q' is q-hierarchical.*

In particular, we obtain the query $Q_{\mathcal{I}}$ by removing atoms from the query $Q_{[m]}$.

Before we prove Claim 9.6 we show how to use the fact to apply Theorem 3.9 on Lemma 9.1 and 9.3 to obtain proof for Theorem 3.12.

Proof for Theorem 3.12. We use in parallel for every $\emptyset \neq \mathcal{I} \subseteq [m]$ the data structures $\mathtt{D}_{\mathcal{I}}$ from Theorem 3.9 for $Q_{\mathcal{I}}$ and D.

Whenever the database receives an update, we update the data structures $\mathtt{D}_{\mathcal{I}}$ in time $\mathit{poly}(Q) \cdot O(\log(\|D\|))$ for all $\emptyset \neq \mathcal{I} \subseteq [m]$. All in all, this can be done in time $\exp(Q) \cdot O(\log(\|D\|))$.

From Theorem 3.9 we have for all $\emptyset \neq \mathcal{I} \subseteq [m]$ corresponding algorithms available (obtained from the assumption of Lemma 9.3). For all $\emptyset \neq \mathcal{I} \subseteq [m]$ the algorithm $\mathtt{A}^{\mathcal{I}}_{\mathsf{enum}}$ has delay $t_d = O(\mathit{poly}(Q))$ (see Theorem 3.9(b)), $\mathtt{A}^{\mathcal{I}}_{\mathsf{count}}$ can be done in time $t_c = O(1)$ (see Theorem 3.9(a)) , $\mathtt{A}^{\mathcal{I}}_{\mathsf{test}}$ can be done in time $t_t = O(\mathit{poly}(Q))$ (see Theorem 3.9(c), $\mathtt{A}^{\mathcal{I}}_{\mathsf{jth}}$ can be done in time $t_j = \mathit{poly}(Q) \cdot O(\log(\|D\|))$ (see Theorem 3.9(e)), $\mathtt{A}^{\mathcal{I}}_{\mathsf{jth\text{-}rev}}$ can be done in time $t_{jr} = \mathit{poly}(Q) \cdot O(\log(\|D\|))$ (see Theorem 3.9(f)), $\mathtt{A}^{\mathcal{I}}_{\mathsf{lex}}$ can be done in time $t_l = \mathit{poly}(Q)$ (see Theorem 3.9(d)).

To enumerate the tuples in $\bigcup_{i=1}^{m} Q_i(D)$, we use the enumeration algorithm from Lemma 9.1 with the enumeration algorithms $\mathtt{A}^{\{i\}}_{\mathsf{enum}}$ and testing algorithms $\mathtt{A}^{\{i\}}_{\mathsf{test}}$ for all $i \in [m]$. Clearly, the tuples can be enumerated with delay $\mathit{poly}(Q)$.

Furthermore, we can apply Lemma 9.3 to obtain an algorithm that upon input of a number $j \in \mathbb{N}$ outputs the tuple $a \in \bigcup_{i=1}^{m} Q_i$ such that a will be the jth tuple in the enumeration in time $\exp(Q)O(\log(\|D\|))$.

This concludes the proof of Theorem 3.12. □

In the remainder of this section we prove Claim 9.6:

Proof of Claim 9.6. We have to show that the following conditions holds:

(1) for all atoms ψ in Q' the set $\mathrm{vars}(\psi)$ forms a directed path in T' that starts from the root and

(2) if $\mathrm{free}(Q') \neq \emptyset$, then $\mathrm{free}(Q')$ is a connected subset in T' that contains the root.

Let ψ be an arbitrary atom in Q'. Clearly, since ψ is an atom in Q and Q is q-hierarchical, the set $\mathrm{vars}(\psi)$ forms a directed path in T (and since $\mathrm{vars}(\psi) \subseteq \mathrm{vars}(Q')$, in particular, in T') that starts from the root. Therefore, Condition (1) holds.

To show that Condition (2) holds, we assume for contradiction that there are (at least) two connected components in free(Q') in T'. Let the free variables z_1 and z_2 be arbitrary variables of the connected components. Since $z_1, z_2 \in V(T') = \text{vars}(Q')$, there are $M, M' \in [m-1]$ such that z_1 is in $\overline{z}_M$ and z_2 is in $\overline{z}_{M'}$. Since Q is q-hierarchical there is a path in T (and in T') from the root to z_1 and from the root to z_2 that only use free variables. These paths only contain variables in $(\{z : z \text{ is in } \overline{z}_M\} \cap \text{free}(Q')) \cup (\{z : z \text{ is in } \overline{z}_{M'}\} \cap \text{free}(Q'))$. In particular there is a path from the root to z_1 and a path from the root to z_2 and therefore, z_1 and z_2 do not belong to different connected components. □

Part II.

Answering FO+MOD Queries under Updates on Bounded Degree Databases

10. Further preliminaries and main results in Part II

This part of the thesis is based on [20]. Here, we consider the task of answering FO+MOD queries under updates on bounded degree databases.

In this section we start with further preliminaries that are necessary for this part. Then the second part's main result is given and at the end the organization of this part is given.

Gaifman graph. The *Gaifman graph* of a σ-db D is the undirected simple graph $G^D = (V, E)$ with vertex set $V := \mathrm{adom}(D)$, where there is an edge between vertices u and v whenever $u \neq v$ and there are $R \in \sigma$ and $(a_1, \ldots, a_{\mathrm{ar}(R)}) \in R^D$ such that $u, v \in \{a_1, \ldots, a_{\mathrm{ar}(R)}\}$. A σ-db D is called *connected* if its Gaifman graph G^D is connected; the *connected components* of D are the connected components of G^D. The *degree* of a database D is the degree of its Gaifman graph G^D, i.e., the maximum number of neighbours of a node of G^D.

Throughout this part of the thesis we fix a number $d \in \mathbb{N}$ and restrict attention to *d-bounded* databases, i.e., to databases of degree at most d.

Further notes for updates. Note that update commands may change the database's degree. In this part of this thesis, we restrict attention to databases of degree at most d. Therefore, when applying an insertion command to a σ-db D of degree $\leqslant d$, the command is carried out only if the resulting database D' still has degree $\leqslant d$; otherwise D remains unchanged and instead of carrying out the insertion command, an error message is returned.

Queries. We fix a countably infinite set **var** of *variables*. In this part of the thesis, we consider the extension FO+MOD of first-order logic FO with modulo-counting quantifiers. For a fixed schema σ, the set FO+MOD$[\sigma]$ is built from atomic formulas of the form $x_1{=}x_2$ and $R(x_1, \ldots, x_{\mathrm{ar}(R)})$, for $R \in \sigma$ and variables $x_1, x_2, \ldots, x_{\mathrm{ar}(R)} \in \mathbf{var}$, and is closed under Boolean connectives $\neg$, $\wedge$, existential first-order quantifiers $\exists x$, and modulo-counting quantifiers $\exists^{i \bmod m} x$, for a variable $x \in \mathbf{var}$ and integers $i, m \in \mathbb{N}$ with $m \geqslant 2$ and $i < m$. The intuitive meaning of a formula of the form $\exists^{i \bmod m} x\, \psi$ is that the number of witnesses x that satisfy ψ is congruent i modulo m. Note that FO+MOD is strictly more expressive than first-order logic without counting quantifiers, since it can express that the number of elements in a unary relation is even, which is not possible to express in FO (cf., e.g., [72]). As usual, $\forall x$, $\vee$, $\rightarrow$, $\leftrightarrow$ will be used as abbreviations when constructing formulas. It will be convenient to add the quantifiers $\exists^{\geqslant m} x$, for $m \in \mathbb{N}_{\geqslant 1}$; a formula of the form $\exists^{\geqslant m} x\, \psi$ expresses that the number of witnesses x which satisfy ψ is $\geqslant m$. Though these quantifiers allow more

succinct definitions, we will treat them as syntactic sugar since they do not increase the expressive power of FO+MOD.

The *quantifier rank* $\mathrm{qr}(\varphi)$ of a FO+MOD-formula φ is the maximum nesting depth of quantifiers that occur in φ. By $\mathrm{free}(\varphi)$ we denote the set of all *free variables* of φ, i.e., all variables x that have at least one occurrence in φ that is not within a quantifier of the form $\exists x$, $\exists^{\geqslant m} x$, or $\exists^{i \bmod m} x$. A *sentence* is a formula φ with $\mathrm{free}(\varphi) = \emptyset$.

An *assignment* for φ is a partial mapping α from **var** to **dom**, where $\mathrm{free}(\varphi) \subseteq \mathrm{dom}(\alpha)$. We write $(D, \alpha) \models \varphi$ to indicate that φ is satisfied when evaluated in D with respect to *active domain semantics* while interpreting every free occurrence of a variable x with the constant $\alpha(x)$. Recall from [2] that "active domain semantics" means that quantifiers are evaluated with respect to the database's active domain. In particular, $(D, \alpha) \models \exists x\, \psi$ if and only if there exists an $a \in \mathrm{adom}(D)$ such that $(D, \alpha\frac{a}{x}) \models \psi$, where $\alpha\frac{a}{x}$ is the assignment α' with $\alpha'(x) = a$ and $\alpha'(y) = \alpha(y)$ for all $y \in \mathrm{dom}(\alpha) \setminus \{x\}$. Accordingly, $(D, \alpha) \models \exists^{\geqslant m} x\ \psi$ if and only if $\big|\big\{\, a \in \mathrm{adom}(D) : (D, \alpha\frac{a}{x}) \models \psi \,\big\}\big| \ \geqslant\ m$, and $(D, \alpha) \models \exists^{i \bmod m} x\ \psi$ if and only if $\big|\big\{\, a \in \mathrm{adom}(D) : (D, \alpha\frac{a}{x}) \models \psi \,\big\}\big| \ \equiv\ i \bmod m$.

A *k-ary* FO+MOD *query of schema σ* is of the form $Q(x_1, \ldots, x_k)$ where $k \in \mathbb{N}$, $Q \in \mathsf{FO{+}MOD}[\sigma]$, and $\mathrm{free}(Q) \subseteq \{x_1, \ldots, x_k\}$. We will often assume that the tuple $(x_1, \ldots, x_k)$ is clear from the context and simply write Q instead of $Q(x_1, \ldots, x_k)$ and $\big(D, (a_1, \ldots, a_k)\big) \models Q$ instead of $\big(D, \frac{a_1,\ldots,a_k}{x_1,\ldots,x_k}\big) \models Q$, where $\frac{a_1,\ldots,a_k}{x_1,\ldots,x_k}$ denotes the assignment α with $\alpha(x_i) = a_i$ for all $i \in [k]$. When evaluated in a σ-db D, the k-ary query $Q(x_1, \ldots, x_k)$ yields the k-ary relation

$$Q(D) \quad := \quad \big\{\, (a_1, \ldots, a_k) \in \mathrm{adom}(D)^k \ :\ \big(D, \tfrac{a_1,\ldots,a_k}{x_1,\ldots,x_k}\big) \ \models\ Q \,\big\}\,.$$

Boolean queries are k-ary queries with $k = 0$. As usual, for Boolean queries we will write $Q(D) = \mathtt{no}$ instead of $Q(D) = \emptyset$, and $Q(D) = \mathtt{yes}$ instead of $Q(D) \neq \emptyset$; and we write $D \models Q$ to indicate that $(D, \alpha) \models Q$ for any assignment α.

Sizes and Cardinalities of Queries. The *size* $\|\sigma\|$ of a schema σ is the sum of the arities of its relation symbols. The size $\|Q\|$ of an FO+MOD query Q of schema σ is the length of Q when viewed as a word over the alphabet $\sigma \cup \mathbf{var} \cup \mathbb{N} \cup \big\{ =, \wedge, \neg, \exists, \mathsf{mod}, \geqslant, (,) \big\} \cup \{,\}$. For a k-ary query $Q(x_1, \ldots, x_k)$ and a σ-db D, the *cardinality of the query result* is the number $|Q(D)|$ of tuples in $Q(D)$. The *cardinality* $|D|$ of a σ-db D is defined as the number of tuples stored in D, i.e., $|D| := \sum_{R \in \sigma} |R^D|$. The *size* $\|D\|$ of D is defined as $\|\sigma\| + |\mathrm{adom}(D)| + \sum_{R \in \sigma} \mathrm{ar}(R) \cdot |R^D|$ and corresponds to the size of a reasonable encoding of D. Throughout this part of the thesis we let $f(Q, d)$ stand for a function of the form

$$f(Q, d) \quad = \quad 2^{d^{2^{O(\|Q\|)}}}. \tag{10.1}$$

For our purposes it will be convenient to assume that $\mathbf{dom} = \mathbb{N}_{\geqslant 1}$.

Our algorithms in this part will take as input a k-ary FO+MOD-query $Q(x_1, \ldots, x_k)$, a parameter d, and an initial σ-db D_0 of degree $\leqslant d$.

Hanf Normal Form for FO+MOD. Our algorithms for evaluating FO+MOD queries rely on a decomposition of FO+MOD queries into *Hanf normal form*. To describe this normal form, we need some more notation.

Two formulas φ and ψ of schema σ are called *d-equivalent* (in symbols: $\varphi \equiv_d \psi$) if for all σ-dbs D *of degree* $\leqslant d$ and all assignments α for φ and ψ in D we have $(D, \alpha) \models \varphi \iff (D, \alpha) \models \psi$.

For a σ-db D and a set $A \subseteq \mathrm{adom}(D)$ we write $D[A]$ to denote the restriction of D to the domain A, i.e., $R^{D[A]} = \left\{\overline{a} \in R^D \,:\, \overline{a} \in A^{\mathrm{ar}(R)}\right\}$, for all $R \in \sigma$. For two σ-dbs D and D', an *isomorphism* $\pi\colon D \to D'$ is a bijection from $\mathrm{adom}(D)$ to $\mathrm{adom}(D')$ with $(b_1, \ldots, b_r) \in R^D \iff (\pi(b_1), \ldots, \pi(b_r)) \in R^{D'}$ for all $R \in \sigma$, for $r := \mathrm{ar}(R)$, and for all $b_1, \ldots, b_r \in \mathrm{adom}(D)$. For two k-tuples $\overline{a} = (a_1, \ldots, a_k)$ and $\overline{a}' = (a'_1, \ldots, a'_k)$ of elements in $\mathrm{adom}(D)$ and $\mathrm{adom}(D')$, resp., we write $\big(D, \overline{a}\big) \cong \big(D', \overline{a}'\big)$ to indicate that there is an isomorphism π from D to D' that maps a_i to a'_i for all $i \in [k]$.

The *distance* $dist^D(a, b)$ between two elements $a, b \in \mathrm{adom}(D)$ is the minimal length (i.e., the number of edges) of a path from a to b in D's Gaifman graph G^D (if no such path exists, we let $dist^D(a, b) = \infty$; note that $dist^D(a, a) = 0$). For $r \geqslant 0$ and $a \in \mathrm{adom}(D)$, the *r-ball* around a in D is the set

$$N_r^D(a) := \left\{b \in \mathrm{adom}(D) \,:\, dist^D(a, b) \leqslant r\right\}.$$

For a σ-db D and a tuple $\overline{a} = (a_1, \ldots, a_k)$ we let $N_r^D(\overline{a}) := \bigcup_{i \in [k]} N_r^D(a_i)$. The *r-neighbourhood* around $\overline{a}$ in D is defined as the σ-db $\mathcal{N}_r^D(\overline{a}) := D[N_r^D(\overline{a})]$. For $r \geqslant 0$ and $k \geqslant 1$, a *type* τ *(over* σ*) with* k *centres and radius* r (for short: *r-type with k centres*) is of the form $(T, \overline{t})$, where T is a σ-db, $\overline{t} \in \mathrm{adom}(T)^k$, and $\mathrm{adom}(T) = N_r^T(\overline{t})$. The elements in $\overline{t}$ are called the *centres* of τ. For a tuple $\overline{a} \in \mathrm{adom}(D)^k$, the *r-type of* $\overline{a}$ *in* D is defined as the r-type with k centres $\big(\mathcal{N}_r^D(\overline{a}), \overline{a}\big)$.

For a given r-type with k centres $\tau = (T, \overline{t})$ it is straightforward to construct a first-order formula $\mathsf{sph}_\tau(\overline{x})$ (depending on r and τ) with k free variables $\overline{x} = (x_1, \ldots, x_k)$ which expresses that the r-type of $\overline{x}$ is isomorphic to τ, i.e., for every σ-db D and all $\overline{a} = (a_1, \ldots, a_k) \in \mathrm{adom}(D)^k$ we have

$$\big(D, \overline{a}\big) \models \mathsf{sph}_\tau(\overline{x}) \iff \big(\mathcal{N}_r^D(\overline{a}), \overline{a}\big) \cong \big(T, \overline{t}\big).$$

The formula $\mathsf{sph}_\tau(\overline{x})$ is called a *sphere-formula* (over σ and $\overline{x}$); the numbers r and k are called *locality radius* and *arity*, resp., of the sphere-formula.

A *Hanf-sentence* (over σ) is a sentence of the form

$$\exists^{\geqslant m} x \;\, \mathsf{sph}_\tau(x) \quad \text{or} \quad \exists^{i \bmod m} x \;\, \mathsf{sph}_\tau(x)\,,$$

where τ is an r-type (over σ) with 1 centre, for some $r \geqslant 0$. The number r is called *locality radius* of the Hanf-sentence. A formula in *Hanf normal form* (over σ) is a Boolean combination[1] of sphere-formulas and Hanf-sentences (over σ). The *locality radius* of a formula ψ in Hanf normal form is the maximum of the locality radii of the Hanf-sentences and the sphere-formulas that occur in ψ. The formula is *d-bounded* if all types τ that occur in sphere-formulas or Hanf-sentences of ψ are d-bounded, i.e., T is of degree $\leqslant d$, where $\tau = (T, \overline{t})$. Our query evaluation algorithms for FO+MOD rely on the following result by Heimberg, Kuske, and Schweikardt [55].

[1] Throughout this thesis, whenever we speak of *Boolean combinations* we mean *finite* Boolean combinations.

Theorem 10.1 ([55]). *There is an algorithm which receives as input a degree bound $d \in \mathbb{N}$ and an $\mathsf{FO}{+}\mathsf{MOD}[\sigma]$-formula φ, and constructs a d-equivalent formula ψ in Hanf normal form (over σ) with the same free variables as φ. For any $d \geqslant 2$, the formula ψ is d-bounded and has locality radius $\leqslant 4^{\mathrm{qr}(\varphi)}$, and the algorithm's runtime is $2^{d^{2^{O(\|\varphi\|+\|\sigma\|)}}}$.*

The first step of all our query evaluation algorithms is to use Theorem 10.1 to transform a given query $Q(\overline{x})$ into a d-equivalent query $\psi(\overline{x})$ in Hanf normal form. The following lemma summarises standard facts, that we will apply at several places throughout Part II to evaluate the sphere-formulas that occur in ψ.

Lemma 10.2. *Let $d \geqslant 2$ and let D be a σ-db of degree $\leqslant d$. Let $r \geqslant 0$, $k \geqslant 1$, and $\overline{a} = (a_1, \ldots, a_k) \in \mathrm{adom}(D)^k$.*

(a) $\left|N_r^D(\overline{a})\right| \leqslant k \sum_{i=0}^{r} d^i \leqslant k d^{r+1}$.

(b) Given D and $\overline{a}$, the r-neighbourhood $\mathcal{N}_r^D(\overline{a})$ can be computed in time $\left(kd^{r+1}\right)^{O(\|\sigma\|)}$.

(c) $\mathcal{N}_r^D(a_1, a_2)$ is connected if and only if $dist^D(a_1, a_2) \leqslant 2r{+}1$.

(d) If $\mathcal{N}_r^D(\overline{a})$ is connected, then $N_r^D(\overline{a}) \subseteq N_{r+(k-1)(2r+1)}^D(a_i)$, for all $i \in [k]$.

(e) Let D' be a σ-db of degree $\leqslant d$ and let $\overline{b} = (b_1, \ldots, b_k) \in \mathrm{adom}(D')^k$. It can be tested in time $(kd^{r+1})^{O(\|\sigma\|+kd^{r+1})} \leqslant 2^{O(\|\sigma\|k^2 d^{2r+2})}$ whether $\left(\mathcal{N}_r^D(\overline{a}), \overline{a}\right) \cong \left(\mathcal{N}_r^{D'}(\overline{b}), \overline{b}\right)$.

Proof. Parts (a)–(d) are straightforward. Concerning Part (e), a brute-force approach is to loop through all mappings from $N_r^D(\overline{a})$ to $N_r^{D'}(\overline{b})$ that map a_i to b_i for every $i \in [k]$ and check whether this mapping is an isomorphism. Each such check can be accomplished in time $n^{O(\|\sigma\|)}$ for $n := kd^{r+1}$, and the number of mappings that have to be checked is $\leqslant n^n$. Thus, the isomorphism test is accomplished in time $n^{O(n+\|\sigma\|)} = (kd^{r+1})^{O(\|\sigma\|+kd^{r+1})}$. □

The time bound stated in part (e) of Lemma 10.2 is obtained by a brute-force approach. When using Luks' polynomial time isomorphism test for bounded degree graphs [75], the time bound of Lemma 10.2(e) can be improved to $\left(kd^{r+1}\right)^{poly(d\|\sigma\|)}$. However, the asymptotic overall runtime of our algorithms for evaluating $\mathsf{FO}{+}\mathsf{MOD}$-queries won't improve when using Luks' algorithm instead of the brute-force isomorphism test of Lemma 10.2(e).

Main Results. The aim of this part is to show the following:

On input of an initial database D_0, and a degree bound d, and a $\mathsf{FO}{+}\mathsf{MOD}$-query Q we can construct in a linear time $f(\|Q\|, d)\|D\|$ preprocessing phase a data structure that can be updated in time $f(\|Q\|, d)$ and allows to

- immediately answer Q on D if Q is a Boolean query (Theorem 11.1),

- test for a given tuple whether if belongs to the result set in time $O(k^2)$ (Theorem 12.5),
- immediately output the number of tuples in the result set $Q(D)$ (Theorem 12.11),
- enumerate the tuples in the result set $Q(D)$ with delay $O(k^2)$ (Theorem 12.14), and
- enumerate the tuples in $Q(D^-) \setminus Q(D^+)$ and $Q(D^+) \setminus Q(D^-)$ with delay $O(k^2)$, where D^- and D^+ denote the database before and after performing the update operation, respectively (Theorem 12.24).

Organization. After some basic definitions in this chapter we obtain a dynamic algorithm for Boolean FO+MOD-queries in Chapter 11. After some preparations for non-Boolean queries in Chapter 12.1, we present the algorithm for testing in Chapter 12.2. In Chapter 12.3 we reduce the task of counting and enumerating FO+MOD-queries in the dynamic setting to the problem of counting and enumerating independent sets in graphs of bounded degree. We use this reduction to provide efficient dynamic counting and enumeration algorithms in Chapter 12.4 and 12.5, respectively. In Chapter 12.6 we generalise this to be able to efficiently enumerate particular subsets of the query result, and we use this in Chapter 12.7 to obtain an efficient dynamic algorithm for enumerating the difference between the old and the new query result. We conclude in Chapter 13.

11. Answering Boolean FO+MOD Queries Under Updates

In [44], Frick and Grohe showed that in the static setting (i.e., without database updates), Boolean FO-queries Q can be answered on databases D of degree $\leqslant d$ in time $f(Q,d)\cdot\|D\|$. Our first main theorem extends their result to FO+MOD-queries and the dynamic setting.

Theorem 11.1. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a Boolean $\mathsf{FO}+\mathsf{MOD}[\sigma]$-query Q and a σ-db D_0 of degree $\leqslant d$, and computes within $t_p = f(Q,d)\cdot\|D_0\|$ preprocessing time a data structure that can be updated in time $t_u = f(Q,d)$ and allows to return the query result $Q(D)$ with answer time $t_{ans} = O(1)$.*

If Q is a d-bounded Hanf-sentence of locality radius r, then $f(Q,d)$ improves to $f(Q,d) = 2^{O(\|\sigma\|d^{2r+2})}$, and the initialisation time is $t_i = O(\|Q\|)$.

The proof will be an easy consequence of Theorem 10.1 and the following lemma.

Lemma 11.2. *There is a dynamic algorithm that receives a schema σ, a number $s \in \mathbb{N}_{\geqslant 1}$, a list $(r_j)_{j\in[s]}$ of non-negative integers, a list $(\rho_j)_{j\in[s]}$ where each ρ_j is a r_j-type with 1 centre (over σ), a degree bound $d \geqslant 2$ and a σ-db D of degree $\leqslant d$. The algorithm computes within $t_i = O(s)$ initialisation time a data structure that can be updated in time $t_u = \sum_{j=1}^{s}\left(d^{r_j+1}\right)^{O(\|\sigma\|+d^{r_j+1})}$. Upon input of a number $j \in [s]$ the algorithm returns within time $O(1)$ the number $\left|\left\{a \in \mathrm{adom}(D) \,:\, (\mathcal{N}_{r_j}^D(a), a) \cong \rho_j\right\}\right|$.*

In particular, the update time t_u is at most $s\cdot 2^{O(\|\sigma\|\cdot d^{2r+2})}$, for $r := \max_{j\in[s]} r_j$.

Proof. For each $j \in [s]$ our data structure will store the number $\mathtt{A}[j]$ of all elements $a \in \mathrm{adom}(D)$ whose r_j-type is isomorphic to ρ_j, i.e., $(\mathcal{N}_{r_j}^D(a), a) \cong \rho_j$. The initialisation for the empty database $D_\emptyset$ lets $\mathtt{A}[j] = 0$ for all $j \in [s]$.

To update our data structure upon a command $\mathsf{update}\, R(a_1,\ldots,a_k)$, for $k = \mathrm{ar}(R)$ and $\mathsf{update} \in \{\mathsf{insert}, \mathsf{delete}\}$, we proceed as follows. The idea is to remove from the data structure the information on all the database elements whose r_j-neighbourhood (for some $j \in [s]$) is affected by the update, and then to recompute the information concerning all these elements on the updated database.

Let D^- be the database before the update is received and let D^+ be the database after the update has been performed. We consider each $j \in [s]$. All elements whose r_j-neighbourhood might have changed, belong to the set $U_j := N_{r_j}^{D'}(\overline{a})$, where $D' := D^+$ if the update command is $\mathsf{insert}\ R(\overline{a})$, and $D' := D^-$ if the update command is $\mathsf{delete}\ R(\overline{a})$.

To remove the old information from $\mathtt{A}[j]$, we compute for each $a \in U_j$ the neighbourhood $T_a := \mathcal{N}_{r_j}^{D^-}(a)$, check whether $(T_a, a) \cong \rho_j$, and if so, decrement the value $\mathtt{A}[j]$.
To recompute the new information for $\mathtt{A}[j]$, we compute for all $a \in U_j$ the neighbourhood $T'_a := \mathcal{N}_{r_j}^{D^+}(a)$, check whether $(T'_a, a) \cong \rho_j$, and if so, increment the value $\mathtt{A}[j]$.

Using Lemma 10.2 we obtain for each $j \in [s]$ that $|U_j| \leqslant kd^{r_j+1}$. For each $a \in U_j$, the neighbourhoods T_a and T'_a can be computed in time $\left(d^{r_j+1}\right)^{O(\|\sigma\|)}$, and testing for isomorphism with ρ_j can be done in time $\left(d^{r_j+1}\right)^{O(\|\sigma\|+d^{r_j+1})}$. Thus, the update of $\mathtt{A}[j]$ is done in time $k{\cdot}\left(d^{r_j+1}\right)^{O(\|\sigma\|+d^{r_j+1})}$. Recall that $k = \mathrm{ar}(R) \leqslant \|\sigma\|$. Hence, the entire update time is $t_u = \sum_{j=1}^{s} \left(d^{r_j+1}\right)^{O(\|\sigma\|+d^{r_j+1})}$. Finally, note that

$$\left(d^{r+1}\right)^{O(\|\sigma\|+d^{r+1})} \leqslant 2^{d^{r+1}\cdot O(\|\sigma\|+d^{r+1})} \leqslant 2^{O(\|\sigma\|\cdot d^{2r+2})}.$$

This completes the proof of Lemma 11.2. □

Proof of Theorem 11.1.
W.l.o.g. we assume that all the symbols of σ occur in Q (otherwise, we remove from σ all symbols that do not occur in Q). In the preprocessing routine, we first use Theorem 10.1 to transform Q into a d-equivalent sentence ψ in Hanf normal form; this takes time $f(Q, d)$. The sentence ψ is a Boolean combination of d-bounded Hanf-sentences (over σ) of locality radius at most $r := 4^{\mathrm{qr}(Q)}$. Let $\rho_1, \ldots, \rho_s$ be the list of all types that occur in ψ. Thus, every Hanf-sentence in ψ is of the form $\exists^{\geqslant k} x\ \mathsf{sph}_{\rho_j}(x)$ or $\exists^{i \bmod m} x\ \mathsf{sph}_{\rho_j}(x)$ for some $j \in [s]$ and $k, i, m \in \mathbb{N}$ with $k \geqslant 1$, $m \geqslant 2$, and $i < m$. For each $j \in [s]$ let r_j be the radius of $\mathsf{sph}_{\rho_j}(x)$. Thus, ρ_j is an r_j-type with 1 centre (over σ).

We use the dynamic data structure provided by Lemma 11.2, and in addition, we also store a Boolean value $\mathtt{Ans}$ where $\mathtt{Ans} = Q(D)$ is the answer of the Boolean query Q on the current database D. This way, the query can be answered in time $O(1)$ by simply outputting $\mathtt{Ans}$.

The initialisation for the empty database $D_\emptyset$ computes $\mathtt{Ans}$ as follows. Every Hanf-sentence of the form $\exists^{\geqslant k} x\ \mathsf{sph}_{\rho_j}(x)$ in ψ is replaced by the Boolean constant false. Every Hanf-sentence of the form $\exists^{i \bmod m} x\ \mathsf{sph}_{\rho_j}(x)$ is replaced by true if $i = 0$ and by false otherwise. The resulting formula, a Boolean combination of the Boolean constants true and false, then is evaluated, and we let $\mathtt{Ans}$ be the obtained result. The entire initialisation takes time at most $t_i = \|\psi\| = f(Q, d) = 2^{d^{2^{O(\|Q\|)}}}$.

To update our data structure upon a command $\mathsf{update}\, R(a_1, \ldots, a_k)$, we first perform the update routine of the data structure provided by Lemma 11.2. Afterwards, we recompute the query answer $\mathtt{Ans}$ as follows. Every Hanf-sentence of the form $\exists^{\geqslant k} x\ \mathsf{sph}_{\rho_j}(x)$ in ψ is replaced by the Boolean constant true if

$$\left|\left\{a \in \mathrm{adom}(D)\ :\ (\mathcal{N}_{r_j}^{D}(a), a) \cong \rho_j\right\}\right| \geqslant k,$$

and by the Boolean constant false otherwise. Every Hanf-sentence of the form $\exists^{i \bmod m} x\ \mathsf{sph}_{\rho_j}(x)$ is replaced by true if

$$\left|\left\{a \in \mathrm{adom}(D) \,:\, (\mathcal{N}_{r_j}^D(a), a) \cong \rho_j\right\}\right| \equiv i \bmod m,$$

and by false otherwise. The resulting formula, a Boolean combination of the Boolean constants true and false, then is evaluated, and we let $\mathtt{Ans}$ be the obtained result. Thus, recomputing $\mathtt{Ans}$ takes time $\mathit{poly}(\|\psi\|)$.

Noting that $r_j \leqslant 4^{\mathrm{qr}(Q)} \leqslant 2^{O(\|Q\|))}$ and $s \leqslant \|\psi\|$, we obtain that the entire update time is

$$t_u \quad \leqslant \quad \sum_{j=1}^{s} \left(d^{r_j+1}\right)^{O(\|\sigma\|+d^{r_j+1})} \;+\; \mathit{poly}(\|\psi\|) \quad \leqslant \quad 2^{d^{2^{O(\|Q\|)}}} \quad = \quad f(Q,d)\,.$$

This completes the proof of Theorem 11.1. □

In [44], Frick and Grohe obtained a matching lower bound for answering Boolean FO-queries of schema $\sigma = \{E\}$ on databases of degree at most $d := 3$ in the static setting. They used the (reasonable) complexity theoretic assumption $\mathsf{FPT} \neq \mathsf{AW}[*]$ and showed that if this assumption is correct, then there is no algorithm that answers Boolean FO-queries Q on σ-dbs D of degree $\leqslant 3$ in time $2^{2^{2^{o(\|Q\|)}}} \cdot \mathit{poly}(\|D\|)$ in the static setting (see Theorem 2 in [44]). As a consequence, the same lower bound holds in the dynamic setting and shows that in Theorem 11.1, the 3-fold exponential dependency on the query size $\|Q\|$ cannot be substantially lowered (unless $\mathsf{FPT} = \mathsf{AW}[*]$):

Corollary 11.3. *Let $\sigma := \{E\}$ and let $d := 3$. If $\mathsf{FPT} \neq \mathsf{AW}[*]$, then there is no dynamic algorithm that receives a Boolean $\mathsf{FO}[\sigma]$-query Q and a σ-db D_0, and computes within $t_p \leqslant f(Q)\cdot \mathit{poly}(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u \leqslant f(Q)$ and allows to return the query result $Q(D)$ with answer time $t_{ans} \leqslant f(Q)$, for a function f with $f(Q) = 2^{2^{2^{o(\|Q\|)}}}$.*

12. Answering non-Boolean FO+MOD Queries Under Updates

12.1. Technical Lemmas on Types and Spheres Useful for Handling Non-Boolean Queries

For our algorithms for evaluating non-Boolean queries it will be convenient to work with a fixed list of representatives of d-bounded r-types, provided by the following lemma.

Lemma 12.1. *There is an algorithm which upon input of a schema σ, a degree bound $d \geqslant 2$, a radius $r \geqslant 0$ and a number $k \geqslant 1$, computes a list $\mathcal{L}_r^{\sigma,d}(k) = \tau_1, \ldots, \tau_\ell$ (for a suitable $\ell \geqslant 1$) of d-bounded r-types with k centres (over σ), such that for every d-bounded r-type τ with k centres (over σ) there is exactly one $i \in [\ell]$ such that $\tau \cong \tau_i$. The algorithm's runtime is $2^{(kd^{r+1})^{O(\|\sigma\|)}}$. Furthermore, upon input of a d-bounded r-type τ with k centres (over σ), the particular $i \in [\ell]$ with $\tau \cong \tau_i$ can be computed in time $2^{(kd^{r+1})^{O(\|\sigma\|)}}$.*

Taking into account the statements of Lemma 10.2 (in particular, the time bound provided by Lemma 10.2(e)), the proof of Lemma 12.1 is straightforward. Throughout the remainder of this part, $\mathcal{L}_r^{\sigma,d}(k)$ will always denote the list provided by Lemma 12.1. The following lemma will be useful for evaluating Boolean combinations of sphere-formulas.

Lemma 12.2. *Let σ be a schema, let $r \geqslant 0$, $k \geqslant 1$, $d \geqslant 2$ and let $\mathcal{L}_r^{\sigma,d}(k) = \tau_1, \ldots, \tau_\ell$. Let $\overline{x} = (x_1, \ldots, x_k)$ be a list of k pairwise distinct variables. For every Boolean combination $\psi(\overline{x})$ of d-bounded sphere-formulas of radius at most r (over σ), there is an $I \subseteq [\ell]$ such that $\psi(\overline{x}) \equiv_d \bigvee_{i \in I} \mathsf{sph}_{\tau_i}(\overline{x})$.*
Furthermore, given $\psi(\overline{x})$, the set I can be computed in time poly$(\|\psi\|) \cdot 2^{(kd^{r+1})^{O(\|\sigma\|)}}$.

Proof. As a first step, we consider each sphere-formula ζ that occurs in ψ and replace it by a d-equivalent disjunction of sphere-formulas $\mathsf{sph}_{\tau_j}(\overline{x})$ with τ_j in $\mathcal{L}_r^{\sigma,d}(k)$: if ζ has arity $k' \leqslant k$ and radius $r' \leqslant r$ and is of the form $\mathsf{sph}_\rho(\overline{x}')$ with $\overline{x}' = (x_{\nu_1}, \ldots, x_{\nu_{k'}})$ for $1 \leqslant \nu_1 < \cdots < \nu_{k'} \leqslant k$ and $\rho = (S, \overline{s})$ with $\overline{s} = (s_1, \ldots, s_{k'})$, then we replace ζ by the formula $\zeta' := \bigvee_{j \in J} \mathsf{sph}_{\tau_j}(\overline{x})$, where J consists of all those $j \in [\ell]$ where for $(T, \overline{t}) = \tau_j$ with $\overline{t} = (t_1, \ldots, t_k)$ and for $\overline{t}' := (t_{\nu_1}, \ldots, t_{\nu_{k'}})$ we have $\big(S, \overline{s}\big) \cong \big(T[N_{r'}^T(\overline{t}')], \overline{t}'\big)$. It is straightforward to see that ζ' and ζ are d-equivalent.

Let ψ_1 be the formula obtained from ψ by replacing each ζ by ζ'. By the Lemmas 12.1 and 10.2, ψ_1 can be constructed in time $O(\|\psi\| \cdot 2^{(kd^{r+1})^{O(\|\sigma\|)}})$. Note that ψ_1 is a Boolean combination of formulas $\mathsf{sph}_{\tau_j}(\overline{x})$ for $j \in [\ell]$.

In the second step, we repeatedly use de Morgan's law to push all $\neg$-symbols in ψ_1 directly in front of sphere-formulas. Afterwards, we replace every subformula of the form $\neg\,\mathsf{sph}_{\tau_j}(\overline{x})$ by the d-equivalent formula $\bigvee_{i\in[\ell]\setminus\{j\}} \mathsf{sph}_{\tau_i}(\overline{x})$. Let ψ_2 be the formula obtained from ψ_1 by these transformations. Constructing ψ_2 from ψ_1 takes time at most $O(\|\psi_1\|) \cdot 2^{(kd^{r+1})^{O(\|\sigma\|)}} = O(\|\psi\| \cdot 2^{(kd^{r+1})^{O(\|\sigma\|)}})$.

In the third step, we eliminate all the $\wedge$-symbols in ψ_2. By the definition of the sphere-formulas $\tau_1, \ldots, \tau_\ell$ we have

$$\mathsf{sph}_{\tau_i}(\overline{x}) \wedge \mathsf{sph}_{\tau_{i'}}(\overline{x}) \quad \equiv_d \quad \begin{cases} \mathsf{sph}_{\tau_i}(\overline{x}), & \text{if } \ i = i' \\ \bot, & \text{if } \ i \neq i' \end{cases} \tag{12.1}$$

where $\bot$ is an unsatisfiable formula. Thus, by the distributive law we obtain for all $m \geqslant 1$ and all $I_1, \ldots, I_m \subseteq [\ell]$ that

$$\begin{aligned} \bigwedge_{j\in[m]} \Big(\bigvee_{i\in I_j} \mathsf{sph}_{\tau_i}(\overline{x}) \Big) \quad &\equiv_d \quad \bigvee_{i_1\in I_1} \cdots \bigvee_{i_m\in I_m} \Big(\mathsf{sph}_{\tau_{i_1}}(\overline{x}) \wedge \cdots \wedge \mathsf{sph}_{\tau_{i_m}}(\overline{x}) \Big) \\ &\equiv_d \quad \bigvee_{i\in I} \mathsf{sph}_{\tau_i}(\overline{x}) \end{aligned}$$

for $I := I_1 \cap \cdots \cap I_m$. We repeatedly use this equivalence during a bottom-up traversal of the syntax-tree of ψ_2 to eliminate all the $\wedge$-symbols in ψ_2. The resulting formula ψ_3 is obtained in time polynomial in the size of ψ_2. Furthermore, ψ_3 is of the desired form $\bigvee_{i\in I} \mathsf{sph}_{\tau_i}(\overline{x})$ for an $I \subseteq [\ell]$. The overall time for constructing ψ_3 and I is $\mathit{poly}(\|\psi\|) \cdot 2^{(kd^{r+1})^{O(\|\sigma\|)}}$. This completes the proof of Lemma 12.2. □

For evaluating a Boolean combination $\psi(\overline{x})$ of sphere-formulas *and Hanf-sentences* on a given σ-db D, an obvious approach is to first consider every Hanf-sentence χ that occurs in ψ, to check if $D \models \chi$, and to replace every occurrence of χ in ψ with true (resp., false) if $D \models \chi$ (resp., $D \not\models \chi$). The resulting formula $\psi'(\overline{x})$ is then transformed into a disjunction $\psi''(\overline{x}) := \bigvee_{i\in I} \mathsf{sph}_{\tau_i}(\overline{x})$ by Lemma 12.2, and the query result $\psi(D) = \psi''(D)$ is obtained as the union of the query results $\mathsf{sph}_{\tau_i}(D)$ for all $i \in I$.

While this works well in the static setting (i.e., without database updates), in the dynamic setting we have to take care of the fact that database updates might change the status of a Hanf-sentence χ in ψ, i.e., an update operation might turn a database D with $D \models \chi$ into a database D' with $D' \not\models \chi$ (and vice versa). Consequently, the formula $\psi''(\overline{x})$ that is equivalent to $\psi(\overline{x})$ on D might be inequivalent to $\psi(\overline{x})$ on D'.

To handle the dynamic setting correctly, at the end of each update step we will use the following lemma, which is an extension of Lemma 12.2 and is proved in a similar way.

Lemma 12.3. *Let σ be a schema. Let $s \geqslant 0$ and let $\chi_1, \ldots, \chi_s$ be arbitrary formulas of schema σ. Let $r \geqslant 0$, $k \geqslant 1$, $d \geqslant 2$ and let $\mathcal{L}_r^{\sigma,d}(k) = \tau_1, \ldots, \tau_\ell$. Let $\overline{x} = (x_1, \ldots, x_k)$ be a list of k pairwise distinct variables. For every Boolean combination $\psi(\overline{x})$ of the formulas $\chi_1, \ldots, \chi_s$ and of d-bounded sphere-formulas of radius at most r (over σ), and for every $J \subseteq [s]$ there is a set $I \subseteq [\ell]$ such that*

$$\psi_J(\overline{x}) \quad \equiv_d \quad \bigvee_{i \in I} \mathsf{sph}_{\tau_i}(\overline{x}),$$

where ψ_J is the formula obtained from ψ by replacing every occurrence of a formula χ_j with true if $j \in J$ and with false if $j \notin J$ (for every $j \in [s]$).
Given ψ and J, the set I can be computed in time poly$(\|\psi\|) \cdot 2^{(kd^{r+1})^{O(\|\sigma\|)}}$.

To evaluate a single sphere-formula $\mathsf{sph}_\tau(\overline{x})$ for a given r-type τ with k centres (over σ), it will be useful to decompose τ into its connected components as follows. Let $\tau = (T, \bar{t})$ with $\bar{t} = (t_1, \ldots, t_k)$. Consider the Gaifman graph G^T of T and let $C_1, \ldots, C_c$ be the vertex sets of the c connected components of G^T. For each connected component C_j of G^T, let $\bar{t}_j$ be the subsequence of $\bar{t}$ consisting of all elements of $\bar{t}$ that belong to C_j, and let k_j be the length of $\bar{t}_j$. Since $(T, \bar{t})$ is an r-type with k centres, we have $T = \mathcal{N}_r^T(\bar{t})$, and thus $c \leqslant k$ and $k_j \geqslant 1$ for all $j \in [c]$. To avoid ambiguity, we make sure that the list $C_1, \ldots, C_c$ is sorted in such a way that for all $j < j'$ we have $i < i'$ for the smallest i with $t_i \in C_j$ and the smallest i' with $t_{i'} \in C_{j'}$.

For each C_j consider the r-type with k_j centres $\rho_j = \big(T[C_j], \bar{t}_j\big)$. Let ν_j be the unique integer such that ρ_j is isomorphic to the ν_j-th element in the list $\mathcal{L}_r^{\sigma,d}(k_j)$, and let τ_{j,ν_j} be the ν_j-th element in this list.

It is straightforward to see that the formula $\mathsf{sph}_\tau(\overline{x})$ is d-equivalent to the formula

$$\mathsf{conn\text{-}sph}_\tau(\overline{x}) \quad := \quad \bigwedge_{j \in [c]} \mathsf{sph}_{\tau_{j,\nu_j}}(\overline{x}_j) \ \wedge \ \bigwedge_{j \neq j'} \neg\, \mathsf{dist}^{k_j,k_{j'}}_{\leqslant 2r+1}(\overline{x}_j, \overline{x}_{j'}), \tag{12.2}$$

where $\overline{x}_j$ is the subsequence of $\overline{x}$ obtained from $\overline{x}$ in the same way as $\bar{t}_j$ is obtained from $\bar{t}$, and $\mathsf{dist}^{k_j,k_{j'}}_{\leqslant 2r+1}(\overline{x}_j, \overline{x}_{j'})$ is a formula of schema σ which expresses that for some variable y in $\overline{x}_j$ and some variable y' in $\overline{x}_{j'}$ the distance between y and y' is $\leqslant 2r{+}1$. I.e., for $\overline{a} = (a_1, \ldots, a_{k_j})$ and $\overline{b} = (b_1, \ldots, b_{k_{j'}})$ we have $(\overline{a}, \overline{b}) \in \mathsf{dist}^{k_j,k_{j'}}_{\leqslant 2r+1}(D) \iff dist^D(\overline{a}; \overline{b}) \leqslant 2r{+}1$, where

$$dist^D(\overline{a}; \overline{b}) \leqslant 2r{+}1 \text{ means that } dist^D(a_i, b_{i'}) \leqslant 2r{+}1 \text{ for some } i \in [k_j] \text{ and } i' \in [k_{j'}]. \tag{12.3}$$

Using the Lemmas 10.2 and 12.1, the following lemma is straightforward.

Lemma 12.4. *There is an algorithm which upon input of a schema σ, numbers $r \geqslant 0$, $k \geqslant 1$ and $d \geqslant 2$ and a r-type τ with k centres (over σ) computes the formula $\mathsf{conn\text{-}sph}_\tau(\overline{x})$, along with the corresponding parameters c and k_j, ν_j, $\overline{x}_j$, τ_{j,ν_j} for all $j \in [c]$.*
The algorithm's runtime is $2^{(kd^{r+1})^{O(\|\sigma\|)}}$.

We define the *signature of* τ w.r.t. r to be the tuple $\mathrm{sgn}_r(\tau)$ built from the parameters c and $\big(k_j, \nu_j, \{\mu \in [k] : x_\mu \text{ belongs to } \overline{x}_j\}\big)_{j\in[c]}$ obtained from the above lemma. The signature $\mathrm{sgn}_r^D(\overline{a})$ of a tuple $\overline{a}$ in a database D w.r.t. radius r is defined as $\mathrm{sgn}_r(\rho)$ for $\rho := \big(\mathcal{N}_r^D(\overline{a}), \overline{a}\big)$. Note that $\overline{a} \in \mathsf{sph}_\tau(D) \iff \mathrm{sgn}_r^D(\overline{a}) = \mathrm{sgn}_r(\tau)$.

12.2. Testing Non-Boolean FO+MOD Queries Under Updates

This section is devoted to the proof of the following theorem.

Theorem 12.5. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a k-ary $\mathsf{FO{+}MOD}[\sigma]$-query $Q(\overline{x})$ (for some $k \in \mathbb{N}$), and a σ-db D_0 of degree $\leqslant d$, and computes within $t_p = f(Q,d) \cdot \|D_0\|$ preprocessing time a data structure that can be updated in time $t_u = f(Q,d)$ and allows to test for any input tuple $\overline{a} \in \mathbf{dom}^k$ whether $\overline{a} \in \varphi(D)$ within testing time $t_t = O(k^2)$.*

For the proof, we use the lemmas provided in Section 12.1 and the following lemma.

Lemma 12.6. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, numbers $r \geqslant 0$ and $k \geqslant 1$, a r-type τ with k centres (over σ), and a σ-db D_0 of degree $\leqslant d$, and computes within $t_p = 2^{(kd^{r+1})^{O(\|\sigma\|)}} \cdot \|D_0\|$ preprocessing time a data structure that can be updated in time $t_u = 2^{(kd^{r+1})^{O(\|\sigma\|)}}$ and allows to test for any input tuple $\overline{a} \in \mathbf{dom}^k$ whether $\overline{a} \in \mathsf{sph}_\tau(D)$ within testing time $t_t = O(k^2)$.*

Proof. The preprocessing routine starts by using Lemma 12.4 to compute the formula $\mathsf{conn\text{-}sph}_\tau(\overline{x})$, along with the according parameters c and k_j, ν_j, $\overline{x}_j$, τ_{j,ν_j} for each $j \in [c]$. This is done in time $2^{(kd^{r+1})^{O(\|\sigma\|)}}$. We let $\mathrm{sgn}_r(\tau)$ be the signature of τ (defined directly after Lemma 12.4). Recall that $\mathsf{conn\text{-}sph}_\tau(\overline{x}) \equiv_d \mathsf{sph}_\tau(\overline{x})$, and recall from equation (12.2) the precise definition of the formula $\mathsf{conn\text{-}sph}_\tau(\overline{x})$. Our data structure will store the following information on the database D:

- the set Γ of all tuples $\overline{b} \in \mathrm{adom}(D)^{k'}$ where $k' \leqslant k$ and $\mathcal{N}_r^D(\overline{b})$ *is connected*, and
- for every $j \in [c]$ and every tuple $\overline{b} \in \Gamma$ of arity k_j, the unique number $\nu_{\overline{b}}$ such that $\rho_{\overline{b}} := \big(\mathcal{N}_r^D(\overline{b}), \overline{b}\big)$ is isomorphic to the $\nu_{\overline{b}}$-th element in the list $\mathcal{L}_r^{\sigma,d}(k_j)$.

We want to store this information in such a way that for any given tuple $\overline{b} \in \mathbf{dom}^{k'}$ it can be checked in time $O(k)$ whether $\overline{b} \in \Gamma$. To ensure this, we use a k'-ary array $\Gamma_{k'}$ [1] that is initialised to 0, and where during update operations the entry $\Gamma_{k'}[\overline{b}]$ is set to 1 for all $\overline{b} \in \Gamma$ of arity k'. In a similar way we can ensure that for any given $j \in [c]$ and any $\overline{b} \in \Gamma$ of arity k_j, the number $\nu_{\overline{b}}$ can be looked up in time $O(k)$.

The **test** routine upon input of a tuple $\overline{a} = (a_1, \ldots, a_k)$ proceeds as follows.

First, we partition $\overline{a}$ into $\overline{a}_1, \ldots, \overline{a}_{c'}$ (for $c' \leqslant k$) such that $C_j := N_r^D(\overline{a}_j)$ for $j \in [c']$ are the connected components of $\mathcal{N}_r^D(\overline{a})$. As in the definition of the formula

[1] This array requires nonlinear but polynomial space

$\mathsf{conn\text{-}sph}_\tau(\overline{x})$, we make sure that this list is sorted in such a way that for all $j < j'$ we have $i < i'$ for the smallest i with $a_i \in C_j$ and the smallest i' with $a_{i'} \in C_{j'}$. All of this can be done in time $O(k^2)$ by first constructing the graph H with vertex set $[k]$ and where there is an edge between vertices i and j iff the tuple (a_i, a_j) belongs to Γ (i. e., $\mathcal{N}_r^D(a_i, a_j)$ is connected), and then computing the connected components of H.

Afterwards, for each $j \in [c']$ we use time $O(k)$ to look up the number $\nu_{\overline{a}_j}$. We then let $\mathrm{sgn}_r^D(\overline{a})$ be the tuple built from c' and $\big(|\overline{a}_j|, \nu_{\overline{a}_j}, \{\mu \in [k] : a_\mu \text{ belongs to } \overline{a}_j\}\big)_{j\in[c']}$. It is straightforward to see that $\overline{a} \in \mathsf{conn\text{-}sph}_\tau(D)$ iff $\mathrm{sgn}_r^D(\overline{a}) = \mathrm{sgn}_r(\tau)$. Therefore, the **test** routine checks whether $\mathrm{sgn}_r^D(\overline{a}) = \mathrm{sgn}_r(\tau)$ and outputs "yes" if this is the case and "no" otherwise. The entire time used by the **test** routine is $t_t = O(k^2)$.

To finish the proof of Lemma 12.6, we have to give further details on the **preprocess** routine and the **update** routine. The **preprocess** routine initialises Γ as the empty set $\emptyset$ and then performs $|D_0|$ update operations to insert all the tuples of D_0 into the data structure. The **update** routine proceeds as follows.

Let D^- be the database before the update is received and let D^+ be the database after the update has been performed. Let the update command be of the form $\mathsf{update}\, R(a_1, \ldots, a_{\mathrm{ar}(R)})$. We let $r' := r + (\mathrm{ar}(R){-}1)(2r{+}1)$. All elements whose r'-neighbourhood might have changed belong to the set $U := N_{r'}^{D'}(\overline{a})$, where $D' := D^+$ if the update command is $\mathsf{insert}\, R(\overline{a})$, and $D' := D^-$ if the update command is $\mathsf{delete}\, R(\overline{a})$.

According to Lemma 10.2(d), all tuples $\overline{b}$ that have to be inserted into or deleted from Γ are built from elements in U. To update the information stored in our data structure, we loop through all tuples of arity $\leqslant k$ that are built from elements in U.

Using Lemma 10.2(a), we obtain that $|U| \leqslant \mathrm{ar}(R){\cdot}d^{r'+1}$. The number of candidate tuples $\overline{b}$ built from elements in U is at most $\big(\mathrm{ar}(R){\cdot}d^{r'+1}\big)^{k+1}$. Using the Lemmas 10.2 and 12.1, it is not difficult to see that the entire update time is at most $t_u = 2^{(kd^{r+1})^{O(\|\sigma\|)}}$. The initialisation time t_i is of the same form, and hence the preprocessing time is as claimed in the lemma. This completes the proof of Lemma 12.6. □

Using Lemma 12.6 and Lemma 12.3, we can show the following.

Lemma 12.7. *Let σ be a schema and let $d \geqslant 2$ be a degree bound. Let $s \geqslant 0$ and let $\chi_1, \ldots, \chi_s$ be arbitrary sentences of schema σ, and assume we have available for each $j \in [s]$ a dynamic algorithm with initialisation time t'_i and update time t'_u that allows to check within answer time t'_{ans} whether or not $D \models \chi_j$ for d-bounded σ-dbs D.*

Then, there is a dynamic algorithm for d-bounded σ-dbs which receives as input numbers $r \geqslant 0$ and $k \geqslant 1$, a tuple $\overline{x} = (x_1, \ldots, x_k)$ of pairwise distinct variables, and a Boolean combination $\psi(\overline{x})$ of the sentences $\chi_1, \ldots, \chi_s$ and of d-bounded sphere-formulas of radius at most r (over σ). Within initialisation time

$$t_i = s(t'_i + t'_{ans}) + \mathit{poly}(\|\psi\|)2^{(kd^{r+1})^{O(\|\sigma\|)}} \tag{12.4}$$

the algorithm builds a data structure that can be updated within time

$$t_u = s(t'_u + t'_{ans}) + \mathit{poly}(\|\psi\|)2^{(kd^{r+1})O(\|\sigma\|)} \tag{12.5}$$

and allows to test for any input tuple $\overline{a} \in \mathrm{adom}(D)^k$ *whether* $\overline{a} \in \psi(D)$ *within testing time*

$$t_t = O(k^2). \tag{12.6}$$

Proof. We use Lemma 12.1 to compute the list $\mathcal{L}_r^{\sigma,d}(k) = \tau_1, \ldots, \tau_\ell$. For each $i \in [\ell]$, we use the dynamic algorithm provided by Lemma 12.6 for $\tau := \tau_i$. Furthermore, for each $j \in [s]$, we use the dynamic algorithm provided by the lemma's assumption for checking whether or not $D \models \chi_j$. In addition to the components used by these dynamic algorithms, our data structure also stores

- the set $J := \{j \in [s] : D \models \chi_j\}$,
- the particular set $I \subseteq [\ell]$ provided by Lemma 12.3 for $\psi(\overline{x})$ and J, and
- the set $K = \{\mathrm{sgn}_r(\tau_i) : i \in I\}$, where for each type τ, $\mathrm{sgn}_r(\tau)$ is the signature of τ defined directly after Lemma 12.4.

The **test** routine upon input of a tuple $\overline{a} = (a_1, \ldots, a_k)$ proceeds in the same way as in the proof of Lemma 12.6 to compute in time $O(k^2)$ the signature $\mathrm{sgn}_r^D(\overline{a})$ of the tuple $\overline{a}$. For every $i \in [\ell]$ we have $\overline{a} \in \mathsf{sph}_{\tau_i}(D) \iff \mathrm{sgn}_r^D(\overline{a}) = \mathrm{sgn}_r(\tau_i)$. Thus, $\overline{a} \in \varphi(D) \iff \mathrm{sgn}_r^D(\overline{a}) \in K$. Therefore, the **test** routine checks whether $\mathrm{sgn}_r^D(\overline{a}) \in K$ and outputs "yes" if this is the case and "no" otherwise. To ensure that this test can be done in time $O(k^2)$, we use an array construction for storing K (similar to the one for storing Γ in the proof of Lemma 12.6).

The **update** routine runs the update routines for all the used dynamic data structures. Afterwards, it recomputes J by calling the **answer** routine for χ_j for all $j \in [s]$. Then, it uses Lemma 12.3 to recompute I. The set K is then recomputed by applying Lemma 12.4 for $\tau := \tau_i$ for all $i \in I$. It is straightforward to verify that the initialisation time t_i, the update time t_u, and the testing time t_t are as claimed by the lemma. □

Theorem 12.5 is now obtained by combining Theorem 10.1, Lemma 12.7, and Theorem 11.1.

Proof of Theorem 12.5.
For $k = 0$, the theorem immediately follows from Theorem 11.1. Consider the case where $k \geqslant 1$. As in the proof of Theorem 11.1, we assume w.l.o.g. that all the symbols of σ occur in φ. We start the preprocessing routine by using Theorem 10.1 to transform $\varphi(\overline{x})$ into a d-equivalent query $\psi(\overline{x})$ in Hanf normal form; this takes time $2^{d^{2^{O(\|\varphi\|)}}}$. The formula ψ is a Boolean combination of d-bounded Hanf-sentences and sphere-formulas (over σ) of locality radius at most $r := 4^{\mathrm{qr}(\varphi)}$, and each sphere-formula is of arity at most k. Let $\chi_1, \ldots, \chi_s$ be the list of all Hanf-sentences that occur in ψ.

From Theorem 11.1 we have available for each $j \in [s]$ a dynamic algorithm with initialisation time $t'_i = O(\max_{j \in [s]} \|\chi_j\|)$ and update time $t'_u = 2^{O(\|\sigma\| d^{2r+2})}$ that allows to check within answer time $t'_{\mathrm{ans}} = O(1)$ whether $D \models \chi_j$ for d-bounded σ-dbs D. The proof of Theorem 12.5 therefore immediately follows from Lemma 12.7. □

12.3. Representing Databases by Coloured Graphs

To obtain dynamic algorithms for counting and enumerating query results, it will be convenient to work with a representation of databases by coloured graphs that is similar to the representation. The main advantage of this representation is that it unveils the combinatorial core of evaluating non-Boolean queries. In Theorem 12.8 we provide a general reduction from evaluating $\mathsf{FO}{+}\mathsf{MOD}[\sigma]$-queries to finding coloured independent sets in coloured graphs, which allows us to focus on this combinatorial graph problem when presenting the algorithms for counting (Section 12.4) and enumeration (Section 12.5).

For defining this representation, let us consider a fixed d-bounded r-type τ with k centres (over a schema σ). Use Lemma 12.4 to compute the formula $\mathsf{conn\text{-}sph}_\tau(\overline{x})$ (for $\overline{x} = (x_1, \dots, x_k)$) and the according parameters c and $k_j, \nu_j, \overline{x}_j, \tau_{j,\nu_j}$, and let $\mathrm{sgn}_r(\tau)$ be the signature of τ. To keep the notation simple, we assume w.l.o.g. that $\overline{x}_1 = x_1, \dots, x_{k_1}$, $\overline{x}_2 = x_{k_1+1}, \dots, x_{k_1+k_2}$ etc.

Recall that $\mathsf{sph}_\tau(\overline{x})$ is d-equivalent to the formula

$$\mathsf{conn\text{-}sph}_\tau(\overline{x}) \quad := \quad \bigwedge_{j\in[c]} \mathsf{sph}_{\tau_{j,\nu_j}}(\overline{x}_j) \;\wedge\; \bigwedge_{j\neq j'} \neg\, \mathsf{dist}^{k_j,k_{j'}}_{\leqslant 2r+1}(\overline{x}_j, \overline{x}_{j'}).$$

To count or enumerate the results of the formula $\mathsf{sph}_\tau(\overline{x})$ we represent the database D by a *c-coloured graph* $\mathcal{G}_D$. Here, a *c-coloured graph* $\mathcal{G}$ is a database of the particular schema

$$\sigma_c \quad := \quad \{E, T_1, \dots, T_c\},$$

where E is a binary relation symbol and $T_1, \dots, T_c$ are unary relation symbols. We define $\mathcal{G}_D$ in such a way that the task of counting or enumerating the results of the query $\mathsf{sph}_\tau(\overline{x})$ on the database D can be reduced to counting or enumerating the results of the query

$$\varphi_c(z_1, \dots, z_c) \quad := \quad \bigwedge_{j\in[c]} T_j(z_j) \;\wedge\; \bigwedge_{j\neq j'} \neg\, E(z_j, z_{j'}) \tag{12.7}$$

on the c-coloured graph $\mathcal{G}_D$. The vertices of $\mathcal{G}_D$ correspond to k_j-tuples over $\mathrm{adom}(D)$ (for some $k_j \in \{k_1, \dots, k_c\} \subseteq [k]$) whose r-neighbourhood is connected; a vertex has colour T_j if its associated tuple $\overline{a}$ is in $\mathsf{sph}_{\tau_{j,\nu_j}}(D)$; and an edge between two vertices indicates that $\mathit{dist}^D(\overline{a}; \overline{b}) \leqslant 2r{+}1$, for their associated tuples $\overline{a}$ and $\overline{b}$. The following theorem allows to translate a dynamic algorithm for counting or enumerating the results of the query $\varphi_c(z_1, \dots, z_c)$ on c-coloured graphs into a dynamic algorithm for counting or enumerating the result of an $\mathsf{FO}{+}\mathsf{MOD}$-query $Q(\overline{x})$ on D.

Theorem 12.8. *Suppose for any $d', c \in \mathbb{N}$ that the counting problem (the enumeration problem) for $\varphi_c(\overline{z})$ on σ_c-dbs of degree at most d' can be solved by a dynamic algorithm with initialisation time $t_i(c, d')$, update time $t_u(c, d')$ and counting time $t_c(c, d')$ (delay $t_d(c, d')$). Then for every schema σ and every $d \geqslant 2$ the following holds. The counting problem (the enumeration problem) for k-ary $\mathsf{FO}{+}\mathsf{MOD}[\sigma]$-queries $Q(\overline{x})$ on σ-dbs of degree at most d can be solved with counting time $O(1)$ (delay $O(\widehat{t_d}{+}k)$), initialisation time $\widehat{t_i} \cdot f(Q, d)$ and update time*

- $(\widehat{t}_u + \widehat{t}_c) \cdot f(Q,d)$ *for the counting problem and*
- $\widehat{t}_u \cdot f(Q,d)$ *for the enumeration problem*

where $\widehat{t}_x = \max_{c=1}^{k} t_x\big(c, d^{2^{O(\|\varphi\|)}}\big)$ *for* $t_x \in \{t_i, t_u, t_c, t_d\}$.

The proof will be obtained as an easy consequence of Theorem 10.1, Theorem 11.1 and the following lemma.

Lemma 12.9. *Suppose that the counting problem (the enumeration problem) for $\varphi_c(\overline{z})$ on σ_c-dbs of degree at most d' can be solved by a dynamic algorithm with initialisation time $t_i(c,d')$, update time $t_u(c,d')$ and counting time $t_c(c,d')$ (delay $t_d(c,d')$).*

Then for every schema σ and every $d \geqslant 2$ the following holds. Let $r \geqslant 0$, $k \geqslant 1$, and fix $d' := d^{2k^2(2r+1)}$ and $\widetilde{t}_x := \max_{c=1}^{k} t_x(c,d')$ for $t_x \in \{t_i, t_u, t_c, t_d\}$.

(1) Let τ be a d-bounded r-type with k centres. The counting problem (the enumeration problem) for $\mathsf{sph}_\tau(\overline{x})$ on σ-dbs of degree at most d can be solved by a dynamic algorithm with counting time $\widetilde{t}_c$ (delay $O(\widetilde{t}_d + k)$), initialisation time $\widetilde{t}_i$, and update time at most $\widetilde{t}_u d^{O(k^2 r + k\|\sigma\|)} + 2^{O(\|\sigma\| k^2 d^{2r+2})}$.

(2) Let $s \geqslant 0$ and let $\chi_1, \ldots, \chi_s$ be arbitrary sentences of schema σ, and assume we have available for each $j \in [s]$ a dynamic algorithm with initialisation time t'_i and update time t'_u that allows to check within answer time t'_{ans} whether or not $D \models \chi_j$ for d-bounded σ-dbs D.

Let $\overline{x} = (x_1, \ldots, x_k)$ be a tuple of pairwise distinct variables, and let $\psi(\overline{x})$ be a Boolean combination of the sentences $\chi_1, \ldots, \chi_s$ and of d-bounded sphere-formulas of radius at most r (over σ).

Then, the counting problem (the enumeration problem) for $\psi(\overline{x})$ on σ-dbs D of degree at most d can be solved by a dynamic algorithm with counting time $O(1)$ (delay $O(\widetilde{t}_d + k)$), initialisation time $s(t'_i + t'_{ans}) + 2^{(kd^{r+1})^{O(\|\sigma\|)}}(poly(\|\psi\|) + \widetilde{t}_i)$, and update time at most

- $s(t'_u + t'_{ans}) + 2^{(k^2 d^{2r+2})^{O(\|\sigma\|)}}(poly(\|\psi\|) + \widetilde{t}_c + \widetilde{t}_u d^{O(k^2 r + k\|\sigma\|)})$ *for the counting problem, and*
- $s(t'_u + t'_{ans}) + 2^{(k^2 d^{2r+2})^{O(\|\sigma\|)}}(poly(\|\psi\|) + t_u d^{O(k^2 r + k\|\sigma\|)})$ *for the enumeration problem.*

Proof. We prove part (1) by a reduction from $\mathsf{conn\text{-}sph}_\tau(\overline{x})$ to φ_c. We use the notation introduced at the beginning of Section 12.3, and we let $\tau_j := \tau_{j,\nu_j}$ for every $j \in [c]$. For a σ-db D of degree at most d we let $\mathcal{G}_D$ be the σ_c-db with

$$\begin{aligned} T_j^{\mathcal{G}_D} &:= \big\{\, v_{\overline{a}} : \overline{a} \in \mathrm{adom}(D)^{k_j} \text{ with } \big(\mathcal{N}_r^D(\overline{a}), \overline{a}\big) \cong \tau_j \,\big\}, \quad \text{for all } j \in [c], \quad \text{and} \\ E^{\mathcal{G}_D} &:= \Big\{\, (v_{\overline{a}}, v_{\overline{b}}) \in V^2 : dist^D(\overline{a}; \overline{b}) \leqslant 2r+1 \,\Big\}, \end{aligned}$$

where $V := \bigcup_{j \in [c]} T_j^{\mathcal{G}_D}$. We will shortly write E and T_j instead of $E^{\mathcal{G}_D}$ and $T_j^{\mathcal{G}_D}$.

Using Lemma 10.2 and the fact that τ_j is connected we obtain that $(v_{\overline{a}}, v_{\overline{b}}) \in E$ iff $\mathcal{N}_r^D(\overline{a}, \overline{b})$ is connected. If $\mathcal{N}_r^D(\overline{a}, \overline{b})$ is connected, then by Lemma 10.2(d) $\overline{b} \in \left(N^D_{r+(|\overline{a}|+|\overline{b}|-1)(2r+1)}(a_1)\right)^{|\overline{b}|}$. It follows that the degree of $\mathcal{G}_D$ is bounded by $d^{2k^2(2r+1)}$. Furthermore, by the definition of $\mathcal{G}_D$ and φ_c we get that $(\overline{a}_1, \ldots, \overline{a}_c) \in \mathsf{sph}_\tau(D) \iff (v_{\overline{a}_1}, \ldots, v_{\overline{a}_c}) \in \varphi_c(\mathcal{G}_D)$, for all tuples $\overline{a}_1, \ldots, \overline{a}_c$ where $\overline{a}_j$ has arity k_j for each $j \in [c]$. As a consequence, $|\mathsf{sph}_\tau(D)| = |\varphi_c(\mathcal{G}_D)|$, and we can therefore use the **count** routine for φ_c on $\mathcal{G}_D$ to count the number of tuples in $\mathsf{sph}_\tau(D)$. Furthermore, by annotating every vertex $v_{\overline{a}}$ with its tuple $\overline{a}$, we can translate every tuple $(v_{\overline{a}_1}, \ldots, v_{\overline{a}_c}) \in \varphi_c(\mathcal{G}_D)$ to $(\overline{a}_1, \ldots, \overline{a}_c)$ in time $O(k)$. Therefore, given an **enumerate** routine for $\varphi_c(\mathcal{G}_D)$ with delay t_d we can produce an enumeration of $\mathsf{sph}_\tau(D)$ with delay $O(t_d + k)$.

It remains to show how to construct and maintain $\mathcal{G}_D$ when the database D is updated. As initialisation for the empty database $D_\emptyset$ we just perform the **init** routine of the dynamic algorithm for $\varphi_c(\overline{z})$ on σ_c-dbs of degree at most d'. The **update** routine of the dynamic algorithm for $\mathsf{sph}_\tau(\overline{x})$ on σ-dbs of degree at most d is provided by the following claim.

Claim 12.10. *If D^+ is obtained from D^- by one update step, then $\mathcal{G}_{D^+}$ can be obtained from $\mathcal{G}_{D^-}$ by $d^{O(k^2 r + k\|\sigma\|)}$ update steps and additional computing time $2^{O(\|\sigma\| k^2 d^{2r+2})}$.*

Proof. Let the update command be of the form $\mathsf{update}\, R(a_1, \ldots, a_{\mathrm{ar}(R)})$ with $\overline{a} = (a_1, \ldots, a_{\mathrm{ar}(R)})$. Let $D' \in \{D^-, D^+\}$ be the database whose relation R contains the tuple $\overline{a}$ (either before deletion or after insertion). Let $r' := r + (k-1)(2r+1)$ and note that all elements in the active domain whose r'-neighbourhood in the database might have changed, belong to the set $U := N_{r'}^{D'}(\overline{a})$.

For every $j \in [c]$ and every tuple $\overline{b}$ of arity at most k of elements in U, we check whether the r-type $\left(\mathcal{N}_r^{D^+}(\overline{b}), \overline{b}\right)$ of $\overline{b}$ is isomorphic to τ_j. Depending on the outcome of this test, we include or exclude $v_{\overline{b}}$ from the relation T_j. Note that it indeed suffices to consider the tuples $\overline{b}$ built from elements in U: The r-type of some tuple $\overline{b}$ is changed by the update command only if $N_r^{D'}(\overline{b})$ contains some element from $\overline{a}$. Furthermore, we only have to consider tuples $\overline{b}$ whose r-neighbourhood $\mathcal{N}_r^{D'}(\overline{b})$ is connected. Using Lemma 10.2(d), we therefore obtain that each component of $\overline{b}$ belongs to $N_{r'}^{D'}(\overline{a}) = U$.

Afterwards, we update the coloured graph's edge relation E. There is an edge $(v_{\overline{b}}, v_{\overline{b}'}) \in E^{D^+}$, if and only if there are $j, j' \in [c]$ such that $v_{\overline{b}} \in C_j$, $v_{\overline{b}'} \in C_{j'}$ and $dist^{D^+}(\overline{b}; \overline{b}') \leqslant 2r+1$. Note that if $dist^{D^+}(\overline{b}; \overline{b}') \leqslant 2r+1$, then there is some component b_i in $\overline{b}$ and some component $b'_{i'}$ in $\overline{b}'$ such that $dist^{D^+}(b_i, b'_{i'}) \leqslant 2r+1$; and in case that $v_{\overline{b}'} \in C_{j'}$, we know that $\mathcal{N}_r^{D^+}(\overline{b}')$ is connected, and hence every component of the tuple $\overline{b}'$ belongs to $N^{D^+}_{r+(k-1)(2r+1)}(b'_{i'}) \subseteq N^{D^+}_{r+k(2r+1)}(b_i) \subseteq N^{D^+}_{r+k(2r+1)}(\overline{b})$. Moreover, for correctly updating the edge relation E it suffices to consider only those pairs of tuples $\overline{b}$ and $\overline{b}'$ where for at least one of the two tuples, at least one component belongs to $N_r^{D'}(\overline{a})$, and hence all components belong to $N_{r'}^{D'}(\overline{a})$, since these are the tuples where the r-neighbourhood or the condition $dist^D(\overline{b}; \overline{b}') \leqslant 2r+1$ might be affected by the database update. Overall, the algorithm proceeds as follows: We compute for all tuples $\overline{b}$ of arity at most k in $N_{r'}^{D'}(\overline{a})$, all tuples $\overline{b}'$ of arity at most k in $N_{r+k(2r+1)}^{D'}(\overline{b})$

(later on, we will call these tuples *candidate tuples*) and check whether (1) there is a $j \in [c]$ such that $v_{\overline{b}} \in T_j$, (2) there is a $j' \in [c]$ such that $v_{\overline{b}'} \in T_{j'}$, and (3) $dist^{D^+}(\overline{b}; \overline{b}') \leqslant 2r+1$. If all three checks return the result "yes", then we insert the tuple $\big(v_{\overline{b}}, v_{\overline{b}'}\big)$ into E, otherwise we remove it from E.

It remains to analyse the runtime of the described update procedure. By Lemma 10.2, $|U| \leqslant \mathrm{ar}(R) d^{r'+1} \leqslant \|\sigma\| d^{r+(k-1)(2r+1)+1} \leqslant d^{O(kr+\lg\|\sigma\|)} \leqslant d^{O(kr+\|\sigma\|)}$. Furthermore, U can be computed in time $\big(\mathrm{ar}(R) d^{r'+1}\big)^{O(\|\sigma\|)} \leqslant d^{O(kr\|\sigma\|+\|\sigma\|^2)}$. The number of tuples $\overline{b}$ that we have to consider is at most $\sum_{i=1}^{k} |U|^i \leqslant |U|^{k+1} \leqslant d^{O(k^2 r+k\|\sigma\|)}$.

For each such $\overline{b}$ we use Lemma 10.2(e) to check in time $2^{O(\|\sigma\| k^2 d^{2r+2})}$ whether the r-type of $\overline{b}$ in D^+ is isomorphic to τ_j, for some $j \in [c]$. In summary, for updating the sets $T_1, \ldots, T_c$ we use at most $c|U|^{k+1} \leqslant d^{O(k^2 r+k\|\sigma\|)}$ calls of the **update** routine of the dynamic algorithm on coloured graphs, and in addition to that we use computation time at most $2^{O(\|\sigma\| k^2 d^{2r+2})}$.

To update the edge relation, we compute for each of the $d^{O(k^2 r+k\|\sigma\|)}$ tuples $\overline{b}$ a list of all its candidate tuples $\overline{b}'$; using Lemma 10.2 this can be done in time $\big(\mathrm{ar}(R) d^{r+k(2r+1)+1}\big)^{O(\|\sigma\|)} \leqslant d^{O(kr\|\sigma\|+\|\sigma\|^2)}$. By Lemma 10.2, the number of candidate tuples is $\leqslant d^{O(k^2 r+k\|\sigma\|)}$. By Lemma 10.2(e), it takes time $2^{O(\|\sigma\| k^2 d^{2r+2})}$ to check if $\big(\mathcal{N}_r^{D^+}(\overline{b}), \overline{b}\big)$ is isomorphic to τ_j and $\big(\mathcal{N}_r^{D^+}(\overline{b}'), \overline{b}'\big)$ is isomorphic to $\tau_{j'}$, for some $j, j' \in [c]$. We can use and maintain an additional array that allows us to check, for any a_i and b_j, in constant time whether $dist^D(a_i, b_j) \leqslant 2r+1$. Overall, we obtain that also the edge relation E can be updated by at most $d^{O(k^2 r+k\|\sigma\|)}$ calls of the **update** routine of the dynamic algorithm on coloured graphs and additional computation time at most $2^{O(\|\sigma\| k^2 d^{2r+2})}$.

This completes the proof of Claim 12.10. □

Finally, the **preprocess** routine of the dynamic algorithm for $\mathsf{sph}_\tau(\overline{x})$ proceeds in the obvious way by first calling the **init** routine for $D_\emptyset$ and then performing $|D_0|$ update steps to insert all the tuples of D_0 into the data structure. This completes the proof of part (1) of Lemma 12.9.

We now turn to the proof of part (2) of Lemma 12.9. We use Lemma 12.1 to compute the list $\mathcal{L}_r^{\sigma,d}(k) = \tau_1, \ldots, \tau_\ell$ within time $2^{(kd^{r+1})^{O(\|\sigma\|)}}$. For each $i \in [\ell]$, we use the dynamic algorithm for $\mathsf{sph}_{\tau_i}(\overline{x})$ provided from the lemma's part (1). Furthermore, for each $j \in [s]$, we use the dynamic algorithm for answering whether or not $D \models \chi_j$, provided by the assumption of part (2) of Lemma 12.9. In addition to the components used by these dynamic algorithms, our data structure also stores

- the set $J := \{j \in [s] : D \models \chi_j\}$ and
- the particular set $I \subseteq [\ell]$ provided by Lemma 12.3 for $\psi(\overline{x})$ and J.

For the case that we want to solve the counting problem, our data structure also stores

- the cardinality $n = |\varphi(D)|$ of the query result.

The **count** routine simply outputs the value n in time $O(1)$.
The **enumerate** routine runs the **enumerate** routine on $\mathsf{sph}_{\tau_i}(D)$ for every $i \in I$. Note that this enumerates, without repetition, all tuples in $\varphi(D)$, because by Lemma 12.3, $\varphi(D)$ is the union of the sets $\mathsf{sph}_{\tau_i}(D)$ for all $i \in I$, and this is a union of pairwise disjoint sets.
The **update** routine runs the update routines for all used dynamic data structures. Afterwards, it recomputes J by calling the **answer** routine for χ_j for all $j \in [s]$. Then, it uses Lemma 12.3 to recompute I. For the case that we want to solve the counting problem, we afterwards recompute the number n by letting $n = \sum_{i \in I} n_i$, where n_i is the result of the **count** routine for τ_i.

By using the statement of part (1) and the assumptions of part (2) of the lemma, it is straightforward to verify that the initialisation time, the update time, and the counting time (the delay) are as claimed by the lemma. □

Theorem 12.8 is now obtained by combining Theorem 10.1, part (2) of Lemma 12.9, and Theorem 11.1.

Proof of Theorem 12.8.
For $k = 0$, the result follows immediately from Theorem 11.1. Consider the case where $k \geqslant 1$. W.l.o.g. we assume that all the symbols of σ occur in Q (otherwise, we remove from σ all symbols that do not occur in Q). We start the preprocessing routine by using Theorem 10.1 to transform $\varphi(\overline{x})$ into a d-equivalent query $\psi(\overline{x})$ in Hanf normal form; this takes time $2^{d^{2^{O(\|Q\|)}}}$. The formula ψ is a Boolean combination of d-bounded Hanf-sentences and sphere-formulas (over σ) of locality radius at most $r := 4^{\mathrm{qr}(\varphi)}$, and each sphere-formula is of arity at most k. Note that for $d' := d^{2k^2(2r+1)}$ as used in the lemma's part (1), it holds that $d' = d^{2^{O(\|\varphi\|)}}$. Let $\chi_1, \ldots, \chi_s$ be the list of all Hanf-sentences that occur in ψ, and note that $s \leqslant 2^{d^{2^{O(\|\varphi\|)}}}$.

From Theorem 11.1 we have available for each $j \in [s]$ a dynamic algorithm with initialisation time $t'_i = O(\max_{j \in [s]} \|\chi_j\|)$ and update time $t'_u = 2^{O(\|\sigma\| d^{2r+2})}$ that allows to check within answer time $t'_{\mathrm{ans}} = O(1)$ whether $D \models \chi_j$ for d-bounded σ-dbs D.

Applying part (2) of Lemma 12.9, we obtain a dynamic algorithm that solves the counting problem (the enumeration problem) for $\varphi(\overline{x})$ on σ-dbs of degree at most d with counting time $O(1)$ (delay $O(\widetilde{t}_d + k) = O(\widehat{t}_d + k)$), initialisation time

$$s(t'_i + t'_{\mathrm{ans}}) + 2^{(kd^{r+1})^{O(\|\sigma\|)}}(\mathit{poly}(\|\psi\|) + \widetilde{t}_i) \quad \leqslant \quad \widehat{t}_i \cdot 2^{d^{2^{O(\|\varphi\|)}}}$$

and update time at most

$$\begin{aligned} & s(t'_u + t'_{\mathrm{ans}}) + 2^{(k^2 d^{2r+2})^{O(\|\sigma\|)}}(\mathit{poly}(\|\psi\|) + \widetilde{t}_c + \widetilde{t}_u d^{O(k^2 r + k\|\sigma\|)}) \\ \leqslant \quad & \big(\widehat{t}_u + \widehat{t}_c\big) \cdot 2^{d^{2^{O(\|\varphi\|)}}} \end{aligned}$$

when dealing with the counting problem and update time at most

$$s(t'_u + t'_{\mathrm{ans}}) + 2^{(k^2 d^{2r+2})^{O(\|\sigma\|)}}(\mathit{poly}(\|\psi\|) + \widetilde{t}_u d^{O(k^2 r + k\|\sigma\|)}) \quad \leqslant \quad \widehat{t}_u \cdot 2^{d^{2^{O(\|\varphi\|)}}}$$

when dealing with the enumeration problem. □

12.4. Counting Results of FO+MOD Queries Under Updates

This section is devoted to the proof of the following theorem.

Theorem 12.11. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a k-ary $\mathsf{FO{+}MOD}[\sigma]$-query $Q(\overline{x})$ (for some $k \in \mathbb{N}$) and a σ-db D_0 of degree $\leqslant d$ and computes within $t_p = f(Q,d) \cdot \|D_0\|$ preprocessing time a data structure that can be updated in time $t_u = f(Q,d)$ and allows to return the cardinality $|Q(D)|$ of the query result within time $O(1)$.*

The theorem follows immediately from Theorem 12.8 and the following dynamic counting algorithm for the query $\varphi_c(\overline{z})$.

Lemma 12.12. *There is a dynamic algorithm that receives a number $c \geqslant 1$, a degree bound $d \geqslant 2$ and a σ_c-db $\mathcal{G}_0$ of degree $\leqslant d$, and computes $|\varphi_c(\mathcal{G})|$ with $d^{O(c^2)}$ initialisation time, $O(1)$ counting time, and $d^{O(c^2)}$ update time.*

Proof. Recall from (12.7) that $\varphi_c(z_1,\ldots,z_c) = \bigwedge_{i\in[c]} T_i(z_i) \wedge \bigwedge_{j\neq j'} \neg E(z_j,z_{j'})$. For all $j,j' \in [c]$ with $j \neq j'$ consider the formula $\theta_{j,j'}(z_1,\ldots,z_c) := E(z_j,z_{j'}) \wedge \bigwedge_{i\in[c]} T_i(z_i)$. Furthermore, let $\alpha(z_1,\ldots,z_c) := \bigwedge_{i\in[c]} T_i(z_i)$. Clearly, for every σ_c-db $\mathcal{G}$ we have

$$\begin{aligned} \alpha(\mathcal{G}) &= T_1^{\mathcal{G}} \times \cdots \times T_c^{\mathcal{G}}, \\ \varphi_c(\mathcal{G}) &= \alpha(\mathcal{G}) \setminus \Big(\bigcup_{j\neq j'} \theta_{j,j'}(\mathcal{G}) \Big), \\ \text{and hence,} \quad |\varphi_c(\mathcal{G})| &= |\alpha(\mathcal{G})| - \Big| \bigcup_{j\neq j'} \theta_{j,j'}(\mathcal{G}) \Big|. \end{aligned}$$

By the *inclusion-exclusion principle* (see Equation 2.2) we obtain for

$$J := \{(j,j') : j,j' \in [c],\ j \neq j'\}$$

that

$$\Big| \bigcup_{j\neq j'} \theta_{j,j'}(\mathcal{G}) \Big| = \sum_{\emptyset\neq K\subseteq J} (-1)^{|K|-1} \Big| \bigcap_{(j,j')\in K} \theta_{j,j'}(\mathcal{G}) \Big| = \sum_{\emptyset\neq K\subseteq J} (-1)^{|K|-1} \big|\varphi_K(\mathcal{G})\big|$$

for the formula $\varphi_K(z_1,\ldots,z_c) := \bigwedge_{i\in[c]} T_i(z_i) \wedge \bigwedge_{(j,j')\in K} E(z_j,z_{j'})$.
Our data structure stores the following values:

- $|T_i^{\mathcal{G}}|$, for each $i \in [c]$, and $n_1 := |\alpha(\mathcal{G})| = \prod_{i\in[c]} |T_i^{\mathcal{G}}|$,
- $|\varphi_K(\mathcal{G})|$, for each $K \subseteq J$ with $K \neq \emptyset$, and
- $n_2 := \sum_{\emptyset\neq K\subseteq J} (-1)^{|K|-1} \big|\varphi_K(\mathcal{G})\big|$ and $n_3 := n_1 - n_2$.

Note that $n_3 = |\varphi_c(\mathcal{G})|$ is the desired size of the query result. Therefore, the **count** routine can answer in time $O(1)$ by just outputting the number n_3.

It remains to show how these values can be initialised and updated during updates of $\mathcal{G}$. The initialisation for the empty graph initialises all the values to 0. In the **update** routine, the values for $|T_i^{\mathcal{G}}|$ and n_1 can be updated in a straightforward way (using time $O(c)$). For each $K \subseteq J$, the update of $|\varphi_K(\mathcal{G})|$ is provided within time $d^{O(c^2)}$ by the following Claim 12.13.

Claim 12.13. *For every $K \subseteq J$, the cardinality $|\varphi_K(\mathcal{G})|$ of a σ_c-db $\mathcal{G}$ of degree at most d can be updated within time $d^{O(c^2)}$ after $d^{O(c^2)} \cdot |\mathcal{G}_0|$ preprocessing time.*

Proof. Consider the directed graph $H := (V, K)$ with vertex set $V := [c]$ and edge set K. Decompose the Gaifman graph of H into its connected components. Let $V_1, \dots, V_s$ be the connected components (for a suitable $s \leqslant c$). For each $i \in [s]$ let $H_i := H[V_i]$ be the induced subgraph of H on V_i. We write K_i to denote the set of edges of H_i. For every $i \in [s]$ let $\ell_i = |V_i|$ and let $t(i,1) < t(i,2) < \cdots < t(i,\ell_i)$ be the ordered list of the vertices in V_i. Consider the query

$$\varphi_{K_i}(z_{t(i,1)}, \dots, z_{t(i,\ell_i)}) \quad := \quad \bigwedge_{j \in V_i} T_j(z_j) \ \wedge \bigwedge_{(j,j') \in K_i} E(z_j, z_{j'}). \tag{12.8}$$

Note that φ_K is the conjunction of the formulas φ_{K_i} for all $i \in [s]$. Since the variables of the formulas φ_{K_i} for $i \in [s]$ are pairwise disjoint, we have $\varphi_K(\mathcal{G}) = \varphi_{K_1}(\mathcal{G}) \times \cdots \times \varphi_{K_s}(\mathcal{G})$ (modulo permutations of the tuples), and thus $|\varphi_K(\mathcal{G})| = \prod_{i \in [s]} |\varphi_{K_i}(\mathcal{G})|$.

For each $i \in [s]$, the value $|\varphi_{K_i}(\mathcal{G})|$ can be computed as follows. For every $v \in \mathrm{adom}(\mathcal{G})$ we consider the set $S_i^v := \big\{(w_{t(i,1)}, \dots, w_{t(i,\ell_i)}) \in \varphi_{K_i}(\mathcal{G}) : w_{t(i,1)} = v\big\}$. Since the Gaifman graph of H_i is connected and has ℓ_i nodes, it follows that each component of every tuple in S_i^v is contained in the $(\ell_i - 1)$-neighbourhood of v in $\mathcal{G}$, and this neighbourhood contains at most d^{ℓ_i} elements. Therefore, $|S_i^v| \leqslant d^{(\ell_i)^2}$, and using breadth-first search starting from v, the set S_i^v can be computed in time $d^{O(c^2)}$. Note that $\varphi_{K_i}(\mathcal{G})$ is the disjoint union of the sets S_i^v for all $v \in \mathrm{adom}(\mathcal{G})$. Therefore, $|\varphi_{K_i}(\mathcal{G})| = \sum_{v \in \mathrm{adom}(\mathcal{G})} |S_i^v|$.

In our data structure we store for every $i \in [s]$ and every $v \in \mathrm{adom}(\mathcal{G})$ the number $\mu_{i,v} = |S_i^v|$. Moreover, for every $i \in [s]$ we store the sum $\mu_i = \sum_{v \in \mathrm{adom}(\mathcal{G})} \mu_{i,v} = |\varphi_{K_i}(\mathcal{G})|$.

The initialisation for the empty σ_c-db $\mathcal{G}_0$ sets all these values to 0. Whenever the colour of a vertex of $\mathcal{G}$ is updated or an edge is inserted or deleted, we update all affected numbers accordingly. Note that a number $\mu_{i,v}$ changes only if v is in the $(c-1)$-neighbourhood around the updated edge or vertex in the graph $\mathcal{G}$. Hence, for at most $2d^c$ vertices v, the numbers $\mu_{i,v}$ are affected by an update, and each of them can be updated in time $d^{O(c^2)}$. Moreover, for each $i \in [s]$, the sum μ_i can be updated in time $O(d^c)$ by subtracting the old value of $\mu_{i,v}$ and adding the new value of $\mu_{i,v}$ for each of the at most $2d^c$ relevant vertices v. Finally, it takes time $O(c)$ to compute the updated value $|\varphi_K(\mathcal{G})| = \prod_{i \in [s]} \mu_i$. The overall time used to produce the update is $d^{O(c^2)}$. □

Once we have available the updated numbers $|\varphi_K(\mathcal{G})|$ for all $K \subseteq J$, the value n_2 can be computed in time $O(|2^J|) \leqslant 2^{O(c^2)}$. And n_3 is then obtained in time $O(1)$. Altogether, performing the **update** routine takes time at most $d^{O(c^2)}$. The **preprocess** routine initialises all values for the empty graph and then uses $|\mathcal{G}_0|$ update steps to insert all the tuples of $\mathcal{G}_0$ into the data structure. This completes the proof of Lemma 12.12. □

12.5. Enumerating Results of FO+MOD Queries Under Updates

In this section we prove — and afterwards improve — the following theorem.

Theorem 12.14. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a k-ary $\mathsf{FO{+}MOD}[\sigma]$-query $Q(\overline{x})$ (for some $k \in \mathbb{N}$) and a σ-db D_0 of degree $\leqslant d$, and computes within $t_p = f(Q,d)\cdot\|D_0\|$ preprocessing time a data structure that can be updated in time $t_u = f(Q,d)$ and allows to enumerate $Q(D)$ with $d^{2^{O(\|\varphi\|)}}$ delay.*

The theorem follows immediately from Theorem 12.8 and the following dynamic enumeration algorithm for the query $\varphi_c(\overline{z})$.

Lemma 12.15. *There is a dynamic algorithm that receives a number $c \geqslant 1$, a degree bound $d \geqslant 2$ and a σ_c-db $\mathcal{G}_0$ of degree $\leqslant d$, and computes within $t_p = d^{poly(c)} \cdot |\mathcal{G}_0|$ preprocessing time a data structure that can be updated in time $d^{poly(c)}$ and allows to enumerate the query result $\varphi_c(\mathcal{G})$ with $O(c^3 d)$ delay.*

Proof. For a σ_c-db $\mathcal{G}$ and a vertex $v \in \mathrm{adom}(\mathcal{G})$ we let $N^{\mathcal{G}}(v)$ be the set of all neighbours of v in $\mathcal{G}$. I.e., $N^{\mathcal{G}}(v)$ is the set of all $w \in \mathrm{adom}(\mathcal{G})$ such that (v,w) or (w,v) belongs to $E^{\mathcal{G}}$.

The underlying idea of the enumeration procedure is the following greedy strategy. We cycle through all vertices $u_1 \in T_1^{\mathcal{G}}$, $u_2 \in T_2^{\mathcal{G}} \setminus N^{\mathcal{G}}(u_1)$, $u_3 \in T_3^{\mathcal{G}} \setminus \big(N^{\mathcal{G}}(u_1) \cup N^{\mathcal{G}}(u_2)\big)$, $\ldots$, $u_c \in T_c^{\mathcal{G}} \setminus \bigcup_{i \leqslant c-1} N^{\mathcal{G}}(u_i)$ and output $(u_1,\ldots,u_c)$. This strategy does not yet lead to a constant delay enumeration, as there might be vertex tuples $(u_1,\ldots,u_i)$ (for $i < c$) that do extend to an output tuple $(u_1,\ldots,u_c)$, but where many possible extensions are checked before this output tuple is encountered. We now show how to overcome this problem and describe an enumeration procedure with $O(c^3 d)$ delay and update time $d^{poly(c)}$.

Note that for every $J \subseteq [c]$ we have $\big|\bigcup_{j\in J} N^{\mathcal{G}}(u_j)\big| \leqslant cd$. Hence, if a set $T_i^{\mathcal{G}}$ contains more than cd elements, we know that *every* considered tuple has an extension $u_i \in T_i^{\mathcal{G}}$ that is not a neighbour of any vertex in the tuple. Let $I := \big\{i \in [c] : \big|T_i^{\mathcal{G}}\big| \leqslant cd\big\}$ be the set of *small* colour classes in $\mathcal{G}$ and to simplify the presentation we assume without loss of generality that $I = \{1,\ldots,s\}$. In our data structure we store the current index set I and the set

$$\mathcal{S} \quad := \quad \big\{\ (u_1,\ldots,u_s) \in T_1^{\mathcal{G}} \times \cdots \times T_s^{\mathcal{G}} \ :\ (u_j,u_{j'}) \notin E^{\mathcal{G}},\ \text{for all } j \neq j' \ \big\} \tag{12.9}$$

of tuples on the small colours. Note that a tuple $(u_1, \ldots, u_s) \in T_1^{\mathcal{G}} \times \cdots \times T_s^{\mathcal{G}}$ extends to an output tuple $(u_1, \ldots, u_c) \in \varphi_c(\mathcal{G})$ if and only if it is contained in $\mathcal{S}$. We store the current sizes of all colours and this enables us to keep the set I of small colours updated. Moreover, as $|\mathcal{S}| \leqslant (cd)^c$, we can update the set $\mathcal{S}$ in time $d^{poly(c)}$ after every update by a brute-force approach. The enumeration procedure is given in Algorithm 18.

Algorithm 18 Enumeration procedure with delay $O(c^3 d)$.

```
1: for all (u_1, ..., u_s) ∈ S do
2:     ENUM(u_1, ..., u_s).
3: Output the end-of-enumeration message EOE.
4:
5: function ENUM(u_1, ..., u_i)
6:     if i = c then
7:         Output the tuple (u_1, ..., u_c).
8:     else
9:         for all u_{i+1} ∈ T_{i+1}^G do
10:            if u_{i+1} ∉ ⋃_{j=1}^{i} N^G(u_j) then ENUM(u_1, ..., u_i, u_{i+1}).
```

It is straightforward to see that this procedure enumerates $\varphi_c(\mathcal{G})$. Let us analyse the delay. Since for all $i > s$ we have $\left|T_i^{\mathcal{G}}\right| > cd$, it follows that every call of $\textsc{Enum}(u_1, \ldots, u_i)$ leads to at least one recursive call of $\textsc{Enum}(u_1, \ldots, u_i, u_{i+1})$. Furthermore, there are at most cd iterations of the loop in line 9 that do *not* lead to a recursive call. As every test in line 10 can be done in time $O(c)$, it follows that the time spans until the first recursive call, between the calls, and after the last call are bounded by $O(c^2 d)$. As the recursion depth is c, the overall delay between two output tuples is bounded by $O(c^3 d)$. □

By using similar techniques as in [38], we obtain the following improved version of Lemma 12.15 where the delay is independent of the degree bound d.

Lemma 12.16. *There is a dynamic algorithm that receives a number $c \geqslant 1$, a degree bound $d \geqslant 2$ and a σ_c-db $\mathcal{G}_0$ of degree $\leqslant d$, and computes within $t_p = d^{poly(c)} \cdot |\mathcal{G}_0|$ preprocessing time a data structure that can be updated in time $d^{poly(c)}$ and allows to enumerate the query result $\varphi_c(\mathcal{G})$ with $O(c^2)$ delay.*

Before proving Lemma 12.16, let us first point out that Lemma 12.16 in combination with Theorem 12.8 directly improves the delay in Theorem 12.14 from $d^{2^{O(\|\varphi\|)}}$ to $O(k^2)$, immediately leading to the following theorem.

Theorem 12.17. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a k-ary $\mathsf{FO{+}MOD}[\sigma]$-query $\varphi(\overline{x})$ (for some $k \in \mathbb{N}$) and a σ-db D_0 of degree $\leqslant d$, and computes within $t_p = f(Q,d) \cdot \|D_0\|$ preprocessing time a data structure that can be updated in time $t_u = f(Q,d)$ and allows to enumerate $\varphi(D)$ with $O(k^2)$ delay.*

The rest of the section is devoted to the proof of Lemma 12.16.

Proof of Lemma 12.16. Consider Algorithm 18, which enumerates $\varphi_c(\mathcal{G})$ with $O(c^3 d)$ delay. To enumerate the tuples with only $O(c^2)$ delay, we replace the loop in lines 9–10 by a precomputed "skip" function that allows to iterate through all elements in $T^{\mathcal{G}}_{i+1} \setminus \bigcup_{j=1}^{i} N^{\mathcal{G}}(u_j)$ with $O(c)$ delay.

For every $i \in [c]$ we store all elements of $T^{\mathcal{G}}_i$ in a doubly linked list and let void be an auxiliary element that appears at the end of the list. We let first_i be the first element in the list and $\mathsf{succ}_i(u)$ the successor of $u \in T^{\mathcal{G}}_i$. We denote by $\leqslant^i$ the linear order induced by this list. We let $\tilde{E}^{\mathcal{G}}$ be the symmetric closure of $E^{\mathcal{G}}$, i.e., $\tilde{E}^{\mathcal{G}} = E^{\mathcal{G}} \cup \big\{(v,u) : (u,v) \in E^{\mathcal{G}}\big\}$. For every $i \in [c]$ we define the function

$$\mathsf{skip}_i(y, V) \quad := \quad \min\Big\{ z \in T^{\mathcal{G}}_i \cup \{\mathsf{void}\} \;:\; y \leqslant^i z \;\text{ and }\; \text{for all } v \in V,\ (v,z) \notin \tilde{E}^{\mathcal{G}} \Big\},$$

which assigns to every $V \subseteq \mathrm{adom}(\mathcal{G})$ with $|V| \leqslant c-1$, and every $y \in T^{\mathcal{G}}_i$ the next node in $T^{\mathcal{G}}_i$ that is not adjacent to any vertex in V.

Using these functions, our improved enumeration algorithm is given in Algorithm 19. Below, we show that we can access the values $\mathsf{skip}_i(y, V)$ in time $O(c)$. By the same analysis as given in the proof of Lemma 12.15 it then follows that Algorithm 19 enumerates $\varphi_c(\mathcal{G})$ with $O(c^2)$ delay.

Algorithm 19 Enumeration procedure with delay $O(c^2)$.

```
1:  for all (u_1, ..., u_s) ∈ S do
2:      ENUM(u_1, ..., u_s).
3:  Output the end-of-enumeration message EOE.
4:
5:  function ENUM(u_1, ..., u_i)
6:      if i = c then
7:          output the tuple (u_1, ..., u_c).
8:      else
9:          y ← skip_{i+1}(first_{i+1}, {u_1, ..., u_i})
10:         while y ≠ void do
11:             ENUM(u_1, ..., u_i, y).
12:             y ← skip_{i+1}(succ_{i+1}(y), {u_1, ..., u_i}).
```

What remains to show is that we can access the values $\mathsf{skip}_i(y, V)$ for all i, y, V in time $O(c)$ and maintain them with $d^{poly(c)}$ update time. At first sight, this is not clear at all, because the domain of skip_i has size $\Omega(|\mathrm{adom}(\mathcal{G})|^c)$. In what follows, we show that for every y, the number of distinct values that $\mathsf{skip}_i(y, V)$ can take is bounded by $d^{poly(c)}$ and that we can store them in a look-up table with update time $d^{poly(c)}$.

To illustrate the main idea, let us start with a simple example. We want to enumerate φ_4 on a coloured graph $\mathcal{H}$ with four vertex colours blue, red, yellow and green (in this order) and analyse the call of $\textsc{Enum}(b, r, y)$, which is supposed to enumerate all green nodes g_i that are not adjacent to any of the nodes b, r and y. The relevant part of $\mathcal{H}$ is depicted in Figure 12.1.

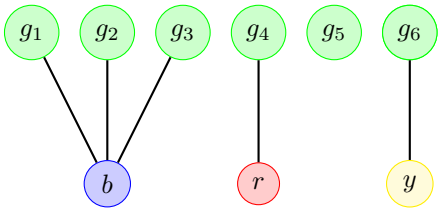


Figure 12.1.: Illustration of the relevant part of graph $\mathcal{H}$.

The enumeration procedure starts by considering the first element g_1 in the list of green vertices, but the first element in the actual output is $g_5 = \mathsf{skip}_4(g_1, \{b, r, y\})$ Therefore, we have to skip the irrelevant vertices $g_1, \ldots, g_4$.

To do this, we want to know the neighbours of the vertices that we skip (b and r in our example) when looking at g_1. For this purpose, we define inductively new sorts of edges $E_4^1 \subseteq E_4^2 \subseteq \cdots$ that connect green vertices $g \in \{g_1, \ldots, g_6\}$ with $\tilde{E}$-neighbours of skipped vertices. In our example, we first have to skip g_1, because it is $\tilde{E}$-connected to b and to be able to handle this, we let E_4^1 be the set of tuples $(g_i, v) \in \tilde{E}^{\mathcal{H}}$ (see Figure 12.2).

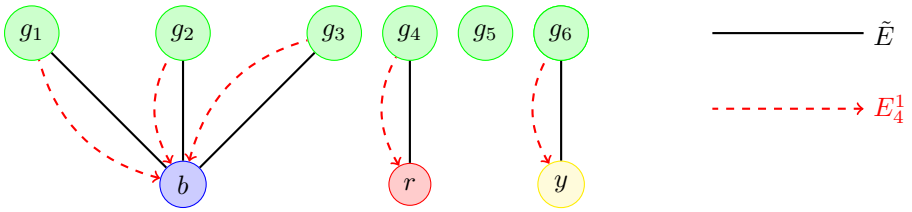


Figure 12.2.: $\tilde{E}$-edges and E_4^1-edges in our example.

After realising that even more vertices (g_2 and g_3) are excluded by b, the next try would be g_4. However, this vertex is excluded by its $\tilde{E}$-neighbour r, so we have to take r into account when computing the skip value for g_1 and indicate this by the E_4^2-edge (g_1, r) (see Figure 12.3). This immediately leads to an inductive definition: E_4^2 contains all pairs of vertices that are already in E_4^1 or connected by a path as shown in Figure 12.4.

The idea outlined above can be formalised as follows. For $i, j \in [c]$, we define inductively the auxiliary edge sets E_i^j:

$$E_i^1 := \left\{ (y,u) : y \in C_i^{\mathcal{G}} \text{ and } (y,u) \in \tilde{E}^{\mathcal{G}} \right\} \quad \text{and}$$

$$E_i^{j+1} := E_i^j \cup \left\{ (y,u) : \begin{array}{c} \text{there are } v,z \text{ with } (y,v) \in E_i^j, \\ (v,z) \in \tilde{E}^{\mathcal{G}}, \text{ and } (\mathsf{succ}_i(z),u) \in \tilde{E}^{\mathcal{G}} \end{array} \right\}$$

Now we define for every $y \in C_i^{\mathcal{G}}$ the set

$$S_i^y := \{ u : (y,u) \in E_i^c \} .$$

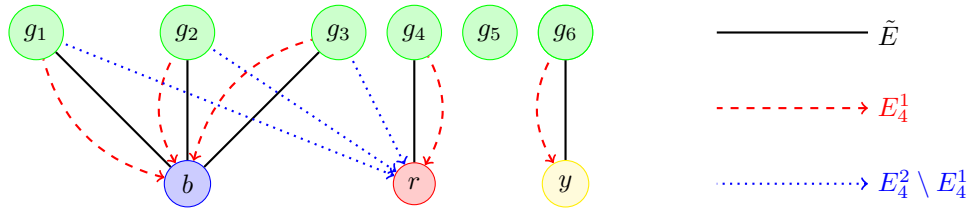


Figure 12.3.: $\tilde{E}$-edges, E_4^1-edges and E_4^2-edges in our example.

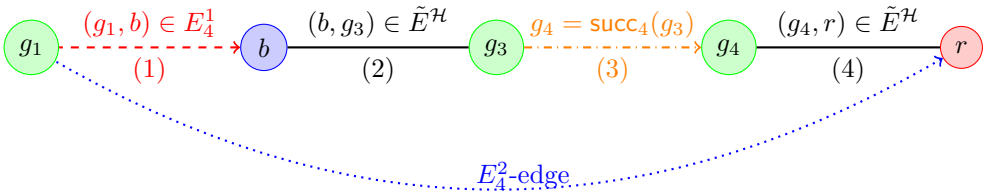


Figure 12.4.: Introducing an E_4^2-edge between g_1 and r.

Note that $|S_i^y| = O(d^{2c})$. The following claim states that the elements of S_i^y are the only ones we need to take into account when computing $\mathsf{skip}_i(y, V)$.

Claim 12.18. *For all $i \leqslant c$, $y \in C_i^{\mathcal{G}} \cup \{\mathsf{void}\}$ and $V \subseteq \mathrm{adom}(\mathcal{G})$ with $|V| \leqslant c-1$ it holds that*

$$\mathsf{skip}_i(y, V) \quad = \quad \mathsf{skip}_i(y, V \cap S_i^y)\,. \tag{12.10}$$

Proof. The proof is identical to the proof of Claim 1 in [38]. For the reader's convenience, the proof is included here. If $c = 1$ or $y = \mathsf{void}$, the claim is trivial. Hence assume that $c \geqslant 2$, $y \neq \mathsf{void}$ and let $z := \mathsf{skip}_i(y, V \cap S_i^y)$. By definition we have $y \leqslant^i z \leqslant^i \mathsf{skip}_i(y, V)$ and therefore we have to show $z \geqslant^i \mathsf{skip}_i(y, V)$, which holds if and only if $(u, z) \notin \tilde{E}^{\mathcal{G}}$ for all $u \in V \setminus S_i^y$. If $z = y$, the claim clearly holds as all $\tilde{E}^{\mathcal{G}}$-neighbours of y are contained in S_i^y. Hence we have $z >^i y$ and let $z' \geqslant^i y$ be the predecessor of z, i.e., $z = \mathsf{succ}_i(z')$. Now assume for contradiction that there is an $u \in V \setminus S_i^y$ such that $(*)$ $(u, z) \in \tilde{E}^{\mathcal{G}}$. Note that since $z' <^i z = \mathsf{skip}_i(y, V \cap S_i^y)$, there is a $v \in V \cap S_i^y$ such that $(**)$ $(v, z') \in \tilde{E}^{\mathcal{G}}$. In the following we show that $(***)$ $(y, v) \in E_i^{c-1}$. Note that this finishes the proof of the claim, as by the definition of E_i^c, the statements $(*)$, $(**)$ and $(***)$ imply that $u \in S_i^y$, contradicting the assumption that $u \in V \setminus S_i^y$.

To show that $(y, v) \in E_i^{c-1}$, let

$$V_j \quad := \quad \left\{ v' \in V \,:\, (y, v') \in E_i^j \right\} \tag{12.11}$$

for all $j \in [c]$. Note that $V_c = V \cap S_i^y$. Furthermore, if there is a $j < c$ with $V_j = V_{j+1}$,

then we have

$$V_j \;=\; V_{j+1} \;=\; \cdots \;=\; V_c \;=\; V \cap S_i^y\,. \tag{12.12}$$

Since $|V| \leqslant c{-}1$ and $u \in V \setminus S_i^y$, we have $|V \cap S_i^y| \leqslant c-2$. In particular, it holds that $V_{c-1} = V \cap S_i^y$. Since $v \in V \cap S_i^y$, it holds that $v \in V_{c-1}$ and thus $(y,v) \in E_i^{c-1}$. □

In our dynamic algorithm we maintain an array that allows random access to the values $\mathsf{skip}_i(y, S')$ for all $y \in C_i^{\mathcal{G}}$ and all $S' \subseteq S_i^y$ of size at most $c{-}1$. By Claim 12.18 we can then compute $\mathsf{skip}_i(y, V)$ by first computing $S' = V \cap S_i^y$ and then looking up $\mathsf{skip}_i(y, S')$. This can be done in time $O(c)$. The next claim states that we can efficiently maintain the sets S_i^y.

Claim 12.19. *There is a data structure that*

1. *stores the elements from the sets S_i^y and all subsets $S' \subseteq S_i^y$ of cardinality at most $c{-}1$,*
2. *allows to test membership in these sets in time $O(1)$ and*
3. *can be updated in time $d^{poly(c)}$ after every update of the form $\mathsf{insert}\ C_i(v)$, $\mathsf{delete}\ C_i(v)$, $\mathsf{insert}\ E(u,v)$, and $\mathsf{delete}\ E(u,v)$.*

Proof. Note that $u \in S_i^y \iff (y,u) \in E_i^c$. We store the edge sets E_i^c for all $i \in [c]$ in adjacency lists and additionally maintain arrays to allow constant-time access to all list entries. In particular, the adjacency list contains for all $y \in V$ a list of elements from S_i^y and access the elements in S_i^y in constant time. Moreover, as the size of S_i^y is bounded by $O(d^{2c})$, the number of subsets $S' \subseteq S_i^y$ of cardinality at most $c{-}1$ is bounded by $O(d^{2c^2})$. Consequently, we can provide constant-time access to all these subsets S'.

On every insertion or deletion of an edge in $E^{\mathcal{G}}$, as well as every insertion or deletion of a vertex in $C_i^{\mathcal{G}}$, we update the edge sets E_i^c according to their inductive definition (see Algorithm 20 for a function computing the E_i^c-edges). To do this efficiently, we use a breadth-first search starting from u and v, for every tuple (u,v) that has changed in relation $E^{\mathcal{G}}$ (or for every u that has changed the membership in $C_i^{\mathcal{G}}$), up to depth $3c$ to identify the relevant nodes that are affected by the change. For all such nodes y we first delete the E_i^c-edges where y is the first component and then we insert the edges E_i^c that are enumerated by ComputeE(y,i,c) in Algorithm 20. It is straightforward to see that the tuples yield in ComputeE(y,i,c) belongs to E_i^c and that this can be done in time $d^{poly(c)}$ as the degree of the edge sets is bounded by $d^{poly(c)}$. For every vertex y where the set S_i^y is recomputed we compute the set $S' \subseteq S_i^y$ of cardinality at most $c-1$. This can be done in time $d^{poly(c)}$.

□

In our data structure we store the values $\mathsf{skip}_i(y, S')$ for every $i \in [c]$, $y \in T_i^{\mathcal{G}}$ and for all sets $S' \subseteq S_i^y$ of cardinality at most $c{-}1$. On every insertion or deletion of an edge, we update the sets S_i^y and their subsets S' of cardinality at most $c{-}1$ and update affected values of $\mathsf{skip}_i(y, S')$. According to Claim 12.19 this can be done in time $d^{poly(c)}$.

Algorithm 20 Compute the sets E_i^j.

```
1: function ComputeE(y, i, j)
2:     if j = 1 then
3:         if y ∈ C_i^G then
4:             for (y, u) ∈ Ẽ^G do
5:                 yield (y, u)
6:     else                                                    ▷ i > 1
7:         for (y, v) in ComputeE(y, i, j − 1) do
8:             yield (y, v)                  ▷ This tuple belongs to E_i^{j−1}
9:             for (v, z) ∈ Ẽ^G do
10:                for (succ_i(z), u) ∈ Ẽ^G do
11:                    yield (y, u)
```

We do the same on updates of the form insert $C_i(v)$ and delete $C_i(v)$, but have to do some additional work, as v might occur in the image of skip-functions. Upon insert $C_i(v)$, we insert v at the beginning of the list C_i. This ensures that existing skip values will not be affected. Afterwards, we compute the set S_i^v and the values $\mathsf{skip}_i(v, S')$ for all $S' \subseteq S_i^v$ of cardinality at most $c-1$. Again, this can be done in time $d^{poly(c)}$.

If we receive the update delete $C_i(v)$, then we have to recompute all skip values $\mathsf{skip}_i(y, S')$ that point to v. Note that (since $\mathcal{G}$ has degree $\leqslant d$) this is only the case for nodes $y \leqslant^i v$ whose distance from v w.r.t. succ_i is at most $(c-1)d$. Hence, it suffices to recompute $\mathsf{skip}_i(y, S')$ for at most $(c-1)d$ vertices y and all $S' \subseteq S_i^y$ of cardinality at most $c-1$. This can be done in time $d^{poly(c)}$. By Claim 12.18, all this suffices to access the value for $\mathsf{skip}_i(y, V)$ in time $O(c)$. This concludes the proof of Lemma 12.16. □

12.6. Refining the enumeration routine

In the previous section we have presented a dynamic algorithm that allows to enumerate with delay $O(k^2)$ the result $Q(D)$ of a k-ary FO+MOD-query $Q(x_1, \ldots, x_k)$ on a database D of degree at most d (see Theorem 12.17). In this section, we generalise this to provide the following functionality. Upon input of a tuple $\overline{a} = (a_1, \ldots, a_\ell) \in \mathbf{dom}^\ell$ for some $\ell \in [k]$, we would like to be able to enumerate all tuples $\overline{b} = (b_1, \ldots, b_k)$ in $Q(D)$ whose first ℓ components coincide with $\overline{a}$. In fact, since we already know that the first ℓ components of $\overline{b}$ coincide with $\overline{a}$, we only output the remaining components $(b_{\ell+1}, \ldots, b_k)$ of $\overline{b}$.

Upon input of the tuple $\overline{a}$, our algorithm spends some *preparation time* t_{prep}, after which it outputs all the desired result tuples with delay t_d.

For formulating this chapter's main result, the following notation will be convenient. Given $k \geqslant 1$ and $\ell \in [k]$, we let $m := k-\ell$. For a tuple $\overline{x} = (x_1, \ldots, x_k)$ of k pairwise distinct variables we let $\overline{z} = (z_1, \ldots, z_\ell) := (x_1, \ldots, x_\ell)$ and $\overline{y} = (y_1, \ldots, y_m) := (x_{\ell+1}, \ldots, x_k)$.

For a k-ary $\mathsf{FO}{+}\mathsf{MOD}[\sigma]$-query $Q(\overline{x}) = Q(\overline{z}, \overline{y})$ and a tuple $\overline{a} = (a_1, \ldots, a_\ell) \in \mathbf{dom}^\ell$ we let $Q(\overline{a}, \overline{y})$ be the m-ary query which, when evaluated on a σ-db D returns the result set

$$\{\, (b_1, \ldots, b_m) \;:\; (a_1, \ldots, a_\ell, b_1, \ldots, b_m) \in Q(D) \,\}\ .$$

This section is devoted to the proof of the following theorem.

Theorem 12.20. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a k-ary $\mathsf{FO}{+}\mathsf{MOD}[\sigma]$-query $Q(\overline{x})$ (for some $k \in \mathbb{N}_{\geqslant 1}$), a number $\ell \in [k]$ and a σ-db D_0 of degree $\leqslant d$, and computes within $t_p = f(Q,d)\cdot\|D_0\|$ preprocessing time a data structure that can be updated in time $t_u = f(Q,d)$ and provides the following functionality: Upon input of an arbitrary tuple $\overline{a} = (a_1, \ldots, a_\ell) \in \mathbf{dom}^\ell$, after $t_{prep} = f(Q,d)$ preparation time it enumerates with delay $O(k^2)$ all result tuples of $Q(\overline{a}, \overline{y})$ on D.*

For proving Theorem 12.20, we will use the following variant of Lemma 12.16, which deals with the queries

$$\varphi_J(z_{j_1}, \ldots, z_{j_c}) \quad := \quad \bigwedge_{j \in J} T_j(z_j) \;\wedge \bigwedge_{j,j' \in J,\ j \neq j'} \neg\, E(z_j, z_j) \tag{12.13}$$

for all sets $J = \{j_1, \ldots, j_c\} \subseteq [2c]$ of cardinality $|J| = c$.

Lemma 12.21. *There is a dynamic algorithm that receives a number $c \geqslant 1$, a degree bound $d \geqslant 2$ and a σ_{2c}-db $\mathcal{G}_0$ of degree $\leqslant d$, and computes within $t_p = d^{poly(c)} \cdot |\mathcal{G}_0|$ preprocessing time a data structure that can be updated in time $d^{poly(c)}$ and provides the following functionality. Upon input of an arbitrary set $J \subseteq [2c]$ of size c, after $d^{poly(c)}$ preparation time it enumerates $\varphi_J(\mathcal{G})$ with delay $O(c^2)$.*

Proof. We use the dynamic algorithm provided by the proof of Lemma 12.16 for input $2c$ instead of c. When receiving an update command, we apply this algorithm's **update** routine.

Recall that the dynamic data structure constructed in the proof of Lemma 12.16 (for $2c$ instead of c) stores the set $I \subseteq [2c]$ of *small* colour classes in $\mathcal{G}$, i.e., $i \in I$ iff $|C_i^{\mathcal{G}}| \leqslant 2cd$.

Upon input of a set J we proceed as follows. Compute the set $I' := I \cap J$ of all small colour classes that are relevant for the query φ_J. For simplicity, let us assume that $I' = \{1, \ldots, s\}$ for some $s \leqslant c$ and $J \setminus I' = \{s{+}1, \ldots, c\}$ (otherwise, we rename the colours accordingly). Within preparation time $d^{poly(c)}$ we can compute the set

$$\mathcal{S} \quad := \quad \Big\{\, (u_1, \ldots, u_s) \in C_1^{\mathcal{G}} \times \cdots \times C_s^{\mathcal{G}} \;:\; \begin{array}{c} (u_j, u_{j'}) \notin E^{\mathcal{G}}, \\ \text{for all } j, j' \in [s] \text{ with } j \neq j' \end{array} \Big\}.$$

To enumerate the query result $\varphi_J(\mathcal{G})$, we then use Algorithm 19 for this set $\mathcal{S}$ (without any changes; in particular, in lines 6 and 7 we don't replace c by $2c$, but keep the value c).

Revisiting the proof of Lemma 12.16 it is straightforward to verify that the resulting dynamic data structure provides the desired functionality within the claimed preparation time and delay. □

Based on the above lemma, we can prove the following variant of Lemma 12.9.

Lemma 12.22. *For $d', c \in \mathbb{N}$ let $t_i(c,d') := (d')^{poly(c)}$, $t_u(c,d') := (d')^{poly(c)}$ and $t_d(c,d') := O(c^2)$.*
For every schema σ and every $d \geqslant 2$ the following holds. Let $r \geqslant 0$, $k \geqslant 1$ and fix $d' := d^{2k^2(2r+1)}$ and $\widetilde{t}_x := \max_{c=1}^{k} t_x(c,d')$ for $t_x \in \{t_i, t_u, t_c, t_d\}$.

(1) Let τ be a d-bounded r-type with k centres and let $\ell \in [k]$. There is a dynamic algorithm which within initialisation time $\widetilde{t}_{\mathsf{init}}$ builds a data structure that can be updated in time at most $\widetilde{t}_{\mathsf{update}} d^{O(k^2 r + k\|\sigma\|)} + 2^{O(\|\sigma\| k^2 d^{2r+2})}$ and provides the following functionality for σ-dbs of degree at most d. Upon input of an arbitrary tuple $\overline{a} \in \mathbf{dom}^\ell$, after $\widetilde{t}_{\mathsf{update}} 2^{O(\|\sigma\| k^2 d^{2r+2})}$ preparation time it enumerates with delay $O(\widetilde{t}_{\mathsf{delay}} + k)$ all result tuples of $\mathsf{sph}_\tau(\overline{a}, \overline{y})$ on D.

(2) Let $s \geqslant 0$ and let $\chi_1, \ldots, \chi_s$ be arbitrary sentences of schema σ, and assume we have available for each $j \in [s]$ a dynamic algorithm with initialisation time t'_i and update time t'_u that allows to check within answer time t'_{ans} whether or not $D \models \chi_j$ for d-bounded σ-dbs D.

Let $\overline{x} = (x_1, \ldots, x_k)$ be a tuple of pairwise distinct variables and let $\psi(\overline{x})$ be a Boolean combination of the sentences $\chi_1, \ldots, \chi_s$ and of d-bounded sphere-formulas of radius at most r (over σ).

There is a dynamic algorithm which within initialisation time

$$s(t'_i + t'_{ans}) + 2^{(kd^{r+1})^{O(\|\sigma\|)}} (poly(\|\psi\|) + \widetilde{t}_{\mathsf{init}}),$$

builds a data structure that can be updated in time

$$s(t'_u + t'_{ans}) + 2^{(k^2 d^{2r+2})^{O(\|\sigma\|)}} (poly(\|\psi\|) + \widetilde{t}_{\mathsf{update}} d^{O(k^2 r + k\|\sigma\|)})$$

and provides the following functionality. Upon input of a tuple $\overline{a} \in \mathbf{dom}^\ell$, after $2^{(k^2 d^{2r+2})^{O(\|\sigma\|)}} \cdot \widetilde{t}_{\mathsf{update}}$ preparation time it enumerates with delay $O(\widetilde{t}_{\mathsf{delay}} + k)$ all result tuples of $\psi(\overline{a}, \overline{y})$ on D.

Proof. For the proof of Part (1) we proceed in a similar way as in the proof of Lemma 12.9(1) and utilise the dynamic algorithm provided by Lemma 12.21.

Upon input of a tuple $\overline{a} = (a_1, \ldots, a_\ell) \in \mathbf{dom}^\ell$ we want to enumerate all tuples $\overline{b} \in \mathbf{dom}^m$ such that $D \models \mathsf{sph}_\tau(\overline{a}, \overline{b})$. We use the same notation as in the proof of Lemma 12.9(1). In particular, recall that we consider the formula

$$\mathsf{conn\text{-}sph}_\tau(\overline{x}) \quad := \quad \bigwedge_{j \in [c]} \mathsf{sph}_{\tau_{j,\nu_j}}(\overline{x}_j) \;\wedge\; \bigwedge_{j \neq j'} \neg\, \mathsf{dist}^{k_j,k_{j'}}_{\leqslant 2r+1}(\overline{x}_j, \overline{x}_{j'})$$

and the coloured graph $\mathcal{G}_D$ of degree at most d' with

$$T_j^{\mathcal{G}_D} := \left\{ v_{\overline{b}} : \overline{b} \in \mathrm{adom}(D)^{k_j} \text{ with } \left(\mathcal{N}_r^D(\overline{b}), \overline{b}\right) \cong \tau_j \right\}, \quad \text{for all } j \in [c], \quad \text{and}$$

$$E^{\mathcal{G}_D} := \left\{ (v_{\overline{b}}, v_{\overline{b}'}) \in V^2 : dist^D(\overline{b}; \overline{b}') \leqslant 2r+1 \right\},$$

where $V := \bigcup_{j\in[c]} T_j^{\mathcal{G}_D}$ and recall that $\tau_j := \tau_{j,\nu_j}$ is a connected r-type for each $j \in [c]$.

Recall that $(x_1, \ldots, x_\ell) = (z_1, \ldots, z_\ell) = \overline{z}$ are the variables the input tuple $\overline{a}$ will be assigned to. W.l.o.g. we assume that there is a number $\kappa \in \{1, \ldots, c\}$ such that the following is true. For each $j \in [\kappa]$, the tuple $\overline{x}_j$ contains at least one of the variables in $\overline{z}$ and for each $j \in [k] \setminus [\kappa]$, the tuple $\overline{x}_j$ contains none of the variables in $\overline{z}$, i.e., it only consists of variables in $\overline{y} = (y_1, \ldots, y_m) = (x_{\ell+1}, \ldots, x_k)$.

For an input tuple $\overline{a}$, we extend $\mathcal{G}_D$ by c further colours $C_{c+1}, \ldots, C_{2c}$ to obtain the following σ_{2c}-db $\mathcal{G}_{D,\overline{a}}$. The edge relation E and the colours $C_1, \ldots, C_c$ of $\mathcal{G}_{D,\overline{a}}$ are the same as those of $\mathcal{G}_D$. For each $j \in \{1, \ldots, \kappa\}$ we let

$$C_{c+j}^{\mathcal{G}_{D,\overline{a}}} \quad := \quad \left\{ v_{\overline{b}} \in C_j^{\mathcal{G}_D} \;:\; \begin{array}{l} \text{for every position } \nu \text{ in } \overline{x}_j \text{ which consists of a vari-} \\ \text{able } z_i \text{ in } \overline{z}, \text{ the entry of } \overline{b} \text{ at position } \nu \text{ is } a_i \end{array} \right\},$$

and for each $j \in \{\kappa{+}1, \ldots, c\}$ we let $C_{c+j}^{\mathcal{G}_{D,\overline{a}}} := \emptyset$.

To enumerate the result tuples of $\mathsf{sph}_\tau(\overline{a}, \overline{y})$ we use the **enumerate** routine provided by Lemma 12.21 upon input

$$J \quad := \quad \{c{+}1, \ldots, c{+}\kappa\} \;\cup\; \{\kappa{+}1, \ldots, c\}$$

to enumerate the set $\varphi_J(\mathcal{G}_{D,\overline{a}})$. Note that according to the definition of the sets $C_{c+j}^{\mathcal{G}_{D,\overline{a}}}$, the set $\varphi_J(\mathcal{G}_{D,\overline{a}})$ contains exactly those tuples in $(v_{\overline{b}_1}, \ldots, v_{\overline{b}_c}) \in \varphi_J(\mathcal{G}_D)$ where for every position ν in some $\overline{x}_j$ that consists of a variable z_i in $\overline{z}$, the entry of $\overline{b}_j$ at position ν is a_i. Therefore, after suitably re-ordering the components of the tuples $(\overline{b}_1, \ldots, \overline{b}_c)$ and dropping the components that correspond to $\overline{a}$, we obain the desired result tuples for $\mathsf{sph}_\tau(\overline{a}, \overline{y})$ on D.

It remains to show how to construct and maintain $\mathcal{G}_{D,\overline{a}}$ when the database is updated or when a new tuple $\overline{a}$ is given as input.

The **update** routine proceeds in exactly the same way as in the proof of Lemma 12.9(1), uses the **update** routine provided by Lemma 12.21, but does not bother to update the colours $C_{c+1}, \ldots, C_{2c}$.

Upon input of a tuple $\overline{a}$, we use the preparation time to first delete all elements from the colours $C_{c+1}, \ldots, C_{2c}$, and then insert all the elements that belong to the sets $C_{c+1}^{\mathcal{G}_{D,\overline{a}}}, \ldots, C_{c+\kappa}^{\mathcal{G}_{D,\overline{a}}}$. The details can be carried out as follows.

Note that according to the definition of $C_{c+j}^{\mathcal{G}_{D,\overline{a}}}$, the following is true for each $j \in [\kappa]$: For every element $v_{\overline{b}}$ in $C_{c+j}^{\mathcal{G}_{D,\overline{a}}}$, some component of the tuple $\overline{b}$ is equal to a component a_i of the tuple $\overline{a}$. Since $\big(\mathcal{N}_r^D(\overline{b}), \overline{b}\big) \cong \tau_j$ and τ_j is a connected r-type, every component of the tuple $\overline{b}$ belongs to $\mathcal{N}_{r'}^D(a_i)$ for $r' := r + (k{-}1)(2r{+}1)$. In particular, from Lemma 10.2 we obtain that $\left|C_{c+j}^{\mathcal{G}_{D,\overline{a}}}\right| \leqslant \left|\mathcal{N}_{r'}^D(a_i)\right|^k \leqslant d^{k^2(2r+1)}$. Thus, the first step of deleting all elements that are still present in C_{c+j} (as an artifact of a previous enumeration request) can be accomplished in time $d^{O(k^2 r)}\widetilde{t}_{\mathsf{update}}$.

Afterwards, we proceed as follows to insert into C_{c+j} all elements that do belong to $C_{c+j}^{\mathcal{G}_{D,\overline{a}}}$. We compute the set $U := \mathcal{N}_{r'}^D(a_i)$ and test for every tuple $\overline{b} \in U^{|\overline{x}_j|}$ whether the following two conditions hold:

(1) The r'-type $(\mathcal{N}_{r'}^{D}(\overline{b}), \overline{b})$ is isomorphic to τ_j, and

(2) for every position ν in $\overline{x}_j$ which consists of a variable z_μ in $\overline{z}$, the entry of $\overline{b}$ at position ν is a_μ.

If both conditions are met, we insert the vertex $v_{\overline{b}}$ into C_{c+j} by using the **update** routine provided by Lemma 12.21. Note that this constructs the correct set $C_{c+j}^{\mathcal{G}_{D,\overline{a}}}$.

To analyse the time needed for this construction, note that by Lemma 10.2, $|U| \leqslant d^{r'+1} \leqslant d^{k(2r+1)} \leqslant d^{O(kr)}$. Furthermore, U can be computed in time $\big(d^{r'+1}\big)^{O(\|\sigma\|)} \leqslant d^{O(kr\|\sigma\|)}$. The number of tuples $\overline{b}$ that we consider is at most $|U|^k \leqslant d^{O(k^2 r)}$. To check if the first condition for $\overline{b}$ is met, we use Lemma 10.2(e) to check in time $2^{O(\|\sigma\|k^2 d^{2r+2})}$ whether the r-type of $\overline{b}$ is isomorphic to τ_j. The second condition can be checked in time $O(k)$. In summary, we compute the sets $C_{c+j}^{\mathcal{G}_{D,\overline{a}}}$ for all $j \in [\kappa]$ in time $d^{O(k^2 r\|\sigma\|)} \cdot 2^{O(\|\sigma\|k^2 d^{2r+2})} \cdot \widetilde{t}_{\mathsf{update}} \leqslant 2^{O(\|\sigma\|k^2 d^{2r+2})} \cdot \widetilde{t}_{\mathsf{update}}$.
This completes the proof of part (1).

Let us now turn to the proof of part (2). The proof can be taken verbatim from the proof of Lemma 12.9(2), with the following changes. Instead of using the dynamic algorithms for $\mathsf{sph}_{\tau_i}(\overline{x})$ provided from Lemma 12.9(1), we now use the dynamic algorithms for $\mathsf{sph}_{\tau_i}(\overline{z}, \overline{y})$ provided by Lemma 12.22(1). Upon input of a tuple $\overline{a} \in \mathbf{dom}^\ell$, we run the preparation phase of the dynamic algorithms for $\mathsf{sph}_{\tau_i}(\overline{a}, \overline{y})$ for all $i \in I$, and afterwards, we loop through all $i \in I$ and enumerate the result tuples of $\mathsf{sph}_{\tau_i}(\overline{a}, \overline{y})$ on D. This completes the proof of Lemma 12.22. □

By using Lemma 12.22(2) instead of Lemma 12.9(2), we obtain the following analogue to Theorem 12.8.

Lemma 12.23. *For $d', c \in \mathbb{N}$ let $t_i(c,d') = (d')^{poly(c)}$, $t_u(c,d') = (d')^{poly(c)}$, and $t_d(c,d') = O(c^2)$. There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a k-ary $\mathsf{FO{+}MOD}[\sigma]$-query $Q(\overline{x})$ (for some $k \in \mathbb{N}_{\geqslant 1}$), and a number $\ell \in [k]$. Within initialisation time $\widehat{t}_{\mathsf{init}} \cdot f(Q,d)$, this algorithm builds a data structure that can be updated in time $\widehat{t}_{\mathsf{update}} \cdot f(Q,d)$ and provides the following functionality for σ-dbs of degree at most d. Upon input of an arbitrary tuple $\overline{a} \in \mathbf{dom}^\ell$, after $\widehat{t}_{\mathsf{update}} \cdot f(Q,d)$ preparation time it enumerates with delay $O(\widehat{t}_{\mathsf{delay}} + k)$ all result tuples of $Q(\overline{a}, \overline{y})$ on D, where $\widehat{t}_x = \max_{c=1}^{k} t_x\big(c, d^{2^{O(\|\varphi\|)}}\big)$ for $t_x \in \{t_i, t_u, t_d\}$.*

Proof. The proof can be taken verbatim from the proof of Theorem 12.8, with the following change. Instead of applying part (2) of Lemma 12.9, we now use the dynamic algorithm provided by part (2) of Lemma 12.22. □

Proof of Theorem 12.20.
The result is an immediate consequence of Lemma 12.23. □

12.7. Enumerating the Difference

In this chapter we consider the update routine called **update_and_report_diff** which, immediately after performing the database update, reports the difference between the new query result and the old query result, i.e., it first enumerates all tuples in $Q(D^+) \setminus Q(D^-)$ (terminated by the end-of-enumeration message `EOE`) and then all tuples in $Q(D^-) \setminus Q(D^+)$ (again, terminated by the message `EOE`), where D^- and D^+ denote the database before and after performing the given update command. This chapter's main result reads as follows.

Theorem 12.24. *There is a dynamic algorithm that receives a schema σ, a degree bound $d \geqslant 2$, a k-ary $\mathsf{FO{+}MOD}[\sigma]$-query $Q(\overline{y})$ (for some $k \in \mathbb{N}$) and a σ-db D_0 of degree $\leqslant d$, and computes within $t_p = f(Q,d)\cdot\|D_0\|$ preprocessing time a data structure providing an* **update_and_report_diff** *routine which, upon input of a command $\mathsf{update}\, R(\overline{a})$ for $\mathsf{update} \in \{\mathsf{insert}, \mathsf{delete}\}$, $R \in \sigma$ and $\overline{a} \in \mathbf{dom}^{\mathrm{ar}(R)}$, updates the data structure within time $t_u = f(Q,d)$ and then enumerates the difference between the new and the old query result with delay $O(k^2)$.*

Proof. We use Theorem 12.20 for the following queries. For each $R \in \sigma$ and each $\mathsf{update} \in \{\mathsf{insert}, \mathsf{delete}\}$ we let $\ell := \mathrm{ar}(R)$ and fix a tuple $\overline{z} = (z_1, \ldots, z_\ell)$ of pairwise distinct variables that do not occur in $Q(\overline{x})$. We construct $(\ell{+}k)$-ary $\mathsf{FO{+}MOD}[\sigma]$-queries

$$Q^+_{\mathsf{update}\, R}(\overline{z}, \overline{x}) \qquad \text{and} \qquad Q^-_{\mathsf{update}\, R}(\overline{z}, \overline{x})$$

such that the following is true for all σ-dbs D and all $\overline{a} \in \mathbf{dom}^\ell$: if D^+ is the σ-db obtained from $D^- \neq D^+$ by performing the update operation $\mathsf{update}\, R(\overline{a})$, then

- the set of result tuples of $Q^+_{\mathsf{update}\, R}(\overline{a}, \overline{x})$ on D^+ is exactly the set $Q(D^+) \setminus Q(D^-)$, and
- the set of result tuples of $Q^-_{\mathsf{update}\, R}(\overline{a}, \overline{x})$ on D^+ is exactly the set $Q(D^-) \setminus Q(D^+)$.

Before constructing these queries let us explain how they can be used to finish the proof of Theorem 12.24. Let Ψ be the set consisting of the queries $Q^+_{\mathsf{update}\, R}$ and $Q^-_{\mathsf{update}\, R}$ for $\mathsf{update} \in \{\mathsf{insert}, \mathsf{delete}\}$ and $R \in \sigma$. We use in parallel for each ψ in Ψ the dynamic algorithm provided by Theorem 12.20. Upon input of an update operation $\mathsf{update}\, R(\overline{a})$, the **update_and_report_diff** routine proceeds as follows. Let $D = D^-$ be the database before executing the update command.

In case that the given update command does not change the database (i.e., the operation intends to delete a tuple that does not belong to the database relation R^D or it intends to insert a tuple that already belongs to R^D or it intends to insert a tuple that would result in a database that exceeds the given degree bound d), then all the data structures remain unchanged and the routine just outputs "`EOE`".

Otherwise, we proceed as follows. First, consider each query ψ in Ψ and perform the **update** routine upon input "$\mathsf{update}\, R(\overline{a})$" of the dynamic algorithm provided by Theorem 12.20 for the query ψ. Let D^+ be the updated database. We then use the functionality provided by Theorem 12.20 to perform in parallel the preparation phase

for the queries $Q^{+}_{\mathsf{update}\,R}(\overline{z},\overline{x})$ and $Q^{-}_{\mathsf{update}\,R}(\overline{z},\overline{x})$ upon input of the tuple $\overline{a}$. Afterwards, we first enumerate the result tuples of $Q^{+}_{\mathsf{update}\,R}(\overline{a},\overline{x})$ on D^{+} and then enumerate the result tuples of $Q^{-}_{\mathsf{update}\,R}(\overline{a},\overline{x})$ on D^{+}.

All that remains to be done to finish the proof of Theorem 12.24 is to construct the queries $Q^{+}_{\mathsf{update}\,R}(\overline{z},\overline{x})$ and $Q^{-}_{\mathsf{update}\,R}(\overline{z},\overline{x})$. The idea is straightforward: we let

$$\begin{aligned} Q^{+}_{\mathsf{update}\,R}(\overline{z},\overline{x}) &:= Q(\overline{x}) \;\wedge\; \neg\, Q^{old}_{\mathsf{update}\,R}(\overline{z},\overline{x}) \qquad \text{and} \\ Q^{-}_{\mathsf{update}\,R}(\overline{z},\overline{x}) &:= Q^{old}_{\mathsf{update}\,R}(\overline{z},\overline{x}) \;\wedge\; \neg\, Q(\overline{x})\,, \end{aligned}$$

where $Q^{old}_{\mathsf{update}\,R}(\overline{z},\overline{x})$ is a formula for which the following is true: if $D^{+} \neq D^{-}$ is obtained from D^{-} by performing the update command $\mathsf{update}\,R(\overline{a})$, then evaluating $Q^{old}_{\mathsf{update}\,R}(\overline{a},\overline{x})$ on D^{+} simulates the evaluation of $Q(\overline{x})$ on D^{-}. A somewhat annoying technical detail in the construction of these formulas is that the semantics of quantifiers takes into account the database's active domain, and the database's active domain might be changed by the update command.

Let us first consider the (easier) case that $\mathsf{update} = \mathsf{insert}$. We let

$$\begin{aligned} \alpha(\overline{z},y) := \exists y_1 \cdots \exists y_\ell \Big(\bigvee_{i=1}^{\ell} y{=}y_i \;\wedge\; R(y_1,\ldots,y_\ell) \;\wedge\; \neg \bigwedge_{i=1}^{\ell} y_i{=}z_i \Big) \;\vee \\ \bigvee_{S\in\sigma\setminus\{R\}} \exists y_1 \cdots \exists y_{\mathrm{ar}(S)} \Big(\bigvee_{i=1}^{\mathrm{ar}(S)} y{=}y_i \;\wedge\; S(y_1,\ldots,y_{\mathrm{ar}(S)}) \Big)\,. \end{aligned}$$

If $D^{+} \neq D^{-}$ is obtained from D^{-} by performing the update command $\mathsf{insert}\,R(\overline{a})$, then the result set of $\alpha(\overline{a},y)$ on D^{+} is exactly the active domain of D^{-}. Therefore, we can choose $Q^{old}_{\mathsf{insert}\,R}(\overline{z},\overline{x})$ to be the query obtained from the input query $Q(\overline{x})$ by replacing every atomic subformula of the form $R(u_1,\ldots,u_\ell)$ with the formula $\big(\, R(u_1,\ldots,u_\ell) \wedge \neg \bigwedge_{i=1}^{\ell} u_i{=}z_i \,\big)$ and by relativising every quantification to a variable y to those y that satisfy $\alpha(\overline{z},y)$, i.e., we replace every subformula of the form $\exists y\,\vartheta$ (or $\exists^{i \bmod m} y\,\vartheta$) with the formula $\exists y\,(\alpha(\overline{z},y) \wedge \vartheta)$ (or $\exists^{i \bmod m} y\,(\alpha(\overline{z},y) \wedge \vartheta)$). It is straightforward to verify that the resulting formula $Q^{old}_{\mathsf{insert}\,R}(\overline{z},\overline{x})$ expresses the desired property.

Let us now turn to the case where $\mathsf{update} = \mathsf{delete}$. The problem here is that $\mathrm{adom}(D^{-})$ contains all the elements in $\overline{a} = (a_1,\ldots,a_\ell)$, while some (or, all) of these elements might be missing in $\mathrm{adom}(D^{+})$, and due to the active domain semantics of FO+MOD, there is no explicit means of enabling quantifiers to range over elements that do not belong to the active domain. To overcome this, let $J \subseteq [\ell]$ be a set of indices such that $|J| = |\,\{a_1,\ldots,a_\ell\}\,|$ and $\{a_1,\ldots,a_\ell\} = \{a_j : j \in J\}$. By induction on the construction of formulas we define for every FO+MOD$[\sigma]$-query $\vartheta(\overline{y})$ that does not contain any of the variables in $\overline{z} = (z_1,\ldots,z_\ell)$ an FO+MOD$[\sigma]$-query $\hat{\vartheta}_J(\overline{z},\overline{y})$ such that the set of result tuples of $\hat{\vartheta}_J(\overline{a},\overline{y})$ on D^{+} is exactly the set $\vartheta(D^{-})$. To achieve this, we proceed as follows:

- if ϑ is of the form $R(y_1,\ldots,y_\ell)$, then $\hat{\vartheta}_J := \big(\, R(y_1,\ldots,y_\ell) \;\vee\; \bigwedge_{i=1}^{\ell} y_i{=}z_i \,\big)$

- if ϑ is of the form $y_1{=}y_2$ or of the form $S(y_1, \ldots, y_{\mathrm{ar}(S)})$ with $S \in \sigma \setminus \{R\}$, then $\hat{\vartheta}_J := \vartheta$
- if ϑ is of the form $\neg\theta$, then $\hat{\vartheta}_J := \neg\hat{\theta}_J$
- if ϑ is of the form $(\theta' \vee \theta'')$, then $\hat{\vartheta}_J := (\widehat{\theta'}_J \vee \widehat{\theta''}_J)$
- if ϑ is of the form $\exists y\, \theta$, then $\hat{\vartheta}_J := \big(\vartheta \vee \bigvee_{j\in J} \hat{\theta}_J \frac{z_j}{y}\big)$, where $\hat{\theta}_J \frac{z_j}{y}$ is the formula obtained from $\hat{\theta}_J$ by replacing every free occurrence of the variable y by the variable z_j
- if ϑ is of the form $\exists^{i \bmod m} y\, \theta$, then $\hat{\vartheta}_J := \bigvee_{(i_1,i_2)\in I} \xi_{(i_1,i_2)}$, where I is the set of all $(i_1, i_2) \in \{0, \ldots, m{-}1\}^2$ with $i_1 + i_2 \equiv i \bmod m$, and

$$\xi_{(i_1,i_2)} \;:=\; \exists^{i_1 \bmod m} y \,\Big(\hat{\theta}_J \wedge \bigwedge_{j\in J} \neg\, y{=}z_j\Big) \;\wedge \bigvee_{\substack{J'\subseteq J\\ |J'|=i_2}} \Big(\bigwedge_{j\in J'} \hat{\theta}_J \tfrac{z_j}{y} \wedge \bigwedge_{j\in J\setminus J'} \neg\hat{\theta}_J \tfrac{z_j}{y}\Big).$$

It is straightforward to verify that the query $\hat{\vartheta}_J(\overline{z}, \overline{y})$ indeed has the desired meaning. Thus, we can choose

$$Q^{old}_{\mathsf{delete}\,R}(\overline{z}, \overline{x}) \;:=\; \bigvee_{J\subseteq[\ell]} \Big(\hat{Q}_J \;\wedge \bigwedge_{\substack{i,j\in J\\ i\neq j}} \neg\, z_i{=}z_j \;\wedge \bigwedge_{i\in[\ell]\setminus J} \bigvee_{j\in J} z_i{=}z_j\Big).$$

This completes the proof of Theorem 12.24. □

13. Conclusion

The main results in the first part show that we can maintain the results of q-hierarchical queries under updates, and they can be tested and counted in constant time and enumerated with constant delay, after a linear time preprocessing, they can be updated in constant time. Furthermore, it was shown that for the testing problem, there is a data structure that can be maintained with constant update time for the more expressive language of t-hierarchical conjunctive queries. Moreover, we showed that if we spend more update time, i.e., logarithmic time in the size of the database, we can also support the jth routine. The results for conjunctive queries have been lifted to so called q-hierarchical (t-hierarchical) unions of conjunctive queries, for which we can support the enumeration and the testing problem under updates and moreover, we considered a subset of q-hierarchical unions of conjunctive queries, the strongly exhaustively q-hierarchial union of conjunctive queries for which we can support the jth problem under updates.

In the papers [16] and [21] it is shown that there are lower bounds for conjunctive queries in the task of dynamic query evaluation. If a self-join free conjunctive query is not q-hierarchical then there is no algorithm with arbitrary preprocessing time and update time sublinear in the size of the databases's active domain, that enumerates sublinear delay, unless the OMv-conjecture (a conjecture on the hardness of online matrix-vector problem [57]) fails. Similar results exist for answering a Boolean conjunctive query and for the counting problem for non-Boolean conjunctive queries. Moreover, there is still no characterisation of the conjunctive queries with aggregates that cannot be maintained under updates. Furthermore, for the jth problem there is no lower bound for unions of conjunctive queries.

Another task we considered is that the preparation of learning a polynomial regression function can be done in constant update time if the training data is taken from a query result of a q-hierarchical query. An obvious future task is to investigate if there are other machine learning models where the training data can be taken from the query, where the data structure is maintained under updates.

In this thesis we considered running times with respect to data complexity. A further future task is to figure out the fine grained complexity of answering conjunctive queries under updates in the combined complexity, i.e., additionally in the size of the query. Moreover, another interesting task is to give an amortized analysis on how the delay depends on the query.

The main results in the second part show that in the dynamic setting (i.e., allowing database updates), the results of k-ary FO+MOD-queries on bounded degree databases can be tested and counted in constant time and enumerated with constant delay, after linear time preprocessing and with constant update time. Here, "constant time" refers to data complexity and is of size $poly(k)$ concerning the delay and the time for testing

and counting. The time for performing a database update is 3-fold exponential in the size of the query and the degree bound, and is worst-case optimal.

The starting point of our algorithms is to decompose the given query into a query in Hanf normal form, using a recent result of [55]. This normal form is only available for the setting with a fixed maximum degree bound d, i.e., the setting considered in this part.

Recently, Kuske and Schweikardt [70] introduced a new kind of Hanf normal form for a variant of *first-order logic with counting* that contains and extends Libkin's logic $\mathsf{FO}(\mathsf{Cnt})$ [72] and Grohe's logic $\mathsf{FO}{+}\mathsf{C}$ [48]. As an application it is shown in [70] that the techniques presented in Part II (Chapter 10 - 12) can be lifted from $\mathsf{FO}{+}\mathsf{MOD}$ to first-order logic with counting terms and numerical predicates $\mathrm{FOC}(\mathbb{P})$.

An obvious future task is to investigate to which extent further query evaluation results that are known for the static setting can be lifted to the dynamic setting. More specifically: are there efficient dynamic algorithms for evaluating (i.e., answering, testing, counting or enumerating) results of first-order queries on other sparse classes of databases (e.g. planar, bounded treewidth, bounded expansion, nowhere dense) or databases of low degree, lifting the "static" results accumulated in [61, 49, 38, 93] to the dynamic setting?

Bibliography

[1] Serge Abiteboul, Marcelo Arenas, Pablo Barceló, Meghyn Bienvenu, Diego Calvanese, Claire David, Richard Hull, Eyke Hüllermeier, Benny Kimelfeld, Leonid Libkin, Wim Martens, Tova Milo, Filip Murlak, Frank Neven, Magdalena Ortiz, Thomas Schwentick, Julia Stoyanovich, Jianwen Su, Dan Suciu, Victor Vianu, and Ke Yi. Research directions for principles of data management (dagstuhl perspectives workshop 16151). *Dagstuhl Manifestos*, 7(1):1–29, 2018.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Georgii M. Adelson-Velskii and Evgenii M. Landis. An algorithm for the organization of information (in russian). In *Doklady Akademii Nauk SSSR*, pages 146:263–266, 1962.

[4] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 89–103, 2019.

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382, 2015.

[6] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[7] Guillaume Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. (Algorithms and complexity of enumeration problems for the evaluation of logical queries)*. PhD thesis, University of Caen Normandy, France, 2009.

[8] Guillaume Bagan, Arnaud, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of the 16th Annual Conference of the EACSL, CSL'07, Lausanne, Switzerland, September 11–15, 2007*, pages 208–222, 2007.

[9] Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *ITA*, 42(1):147–164, 2008.

[10] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[11] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[12] Daniel Barbará and Tomasz Imielinski. Sleepers and workaholics: Caching strategies in mobile environments. *VLDB J.*, 4(4):567–602, 1995.

[13] Michael Benedikt and H. Jerome Keisler. Expressive power of unary counters. In *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, pages 291–305, 1997.

[14] Michael Benedikt and Leonid Libkin. Exact and approximate aggregation in constraint query. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, pages 102–113, 1999.

[15] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. *CoRR*, abs/1702.06370, 2017.

[16] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS'17, Chicago, IL, USA, May 14–19, 2017*, pages 303–318, 2017. Full version available at `http://arxiv.org/abs/1702.06370`.

[17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *CoRR*, abs/1702.08764, 2017.

[18] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory, ICDT 2017, March 21–24, 2017, Venice, Italy*, volume 68 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

[19] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. *CoRR*, abs/1709.10039, 2017.

[20] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *ACM Trans. Database Syst.*, 43(2):7:1–7:32, 2018.

[21] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, pages 8:1–8:19, 2018.

[22] Henrik Björklund, Wouter Gelade, and Wim Martens. Incremental xpath evaluation. *ACM Trans. Database Syst.*, 35(4):29:1–29:43, 2010.

[23] Johann Brault-Baron. *De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre. (The relevance of the list: propositional logic and complexity of the first order).* PhD thesis, University of Caen Normandy, France, 2013.

[24] Andrei A. Bulatov, Víctor Dalmau, Martin Grohe, and Dániel Marx. Enumerating homomorphisms. *J. Comput. Syst. Sci.*, 78(2):638–650, 2012.

[25] Peter J Cameron. *Combinatorics: topics, techniques, algorithms.* Cambridge University Press, 1994.

[26] Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 134–148, 2019.

[27] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 190–200, 1995.

[28] Hubie Chen and Stefan Mengel. A trichotomy in the complexity of counting answers to conjunctive queries. In *Proceedings of the 18th International Conference on Database Theory, ICDT'15, Brussels, Belgium, March 23–27, 2015*, pages 110–126, 2015.

[29] Hubie Chen and Stefan Mengel. Counting answers to existential positive queries: A complexity classification. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS'16, San Francisco, CA, USA, June 26 – July 01, 2016*, pages 315–326, 2016.

[30] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[31] Sara Cohen. Containment of aggregate queries. *SIGMOD Record*, 34(1):77–85, 2005.

[32] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[33] Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theoretical Computer Science, Volume 329, Issues 1–3*, pages 315–323, December 2004.

[34] Nilesh Dalvi and Dan Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *Proceedings of the 26th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'07, Beijing, China, June 11–13, 2007*, pages 293–302. ACM, 2007.

[35] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 61–72, 2002.

[36] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4), 2007.

[37] Arnaud Durand and Stefan Mengel. Structural tractability of counting of solutions to conjunctive queries. *Theory of Computing Systems, Volume 57, Issue 4*, pages 1202–1249, November 2015. Full version of an ICDT 2013 paper.

[38] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating answers to first-order queries over databases of low degree. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22–27, 2014*, pages 121–131, 2014.

[39] Arnaud Durand and Yann Strozecki. Enumeration complexity of logical query problems with second-order variables. In *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings*, pages 189–202, 2011.

[40] Kousha Etessami and Neil Immerman. Tree canonization and transitive closure. *Inf. Comput.*, 157(1-2):2–24, 2000.

[41] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On the complexity of processing massive, unordered, distributed data. *CoRR*, abs/cs/0611108, 2006.

[42] Robert Fink and Dan Olteanu. Dichotomies for queries with negation in probabilistic databases. *ACM Trans. Database Syst.*, 41(1):4:1–4:47, 2016.

[43] Peter Flach. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press, 2012.

[44] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Logic*, 130(1–3):3–31, 2004.

[45] Gianluigi Greco and Francesco Scarcello. Structural tractability of enumerating CSP solutions. *Constraints*, 18(1):38–74, 2013.

[46] Gianluigi Greco and Francesco Scarcello. Counting solutions to conjunctive queries: structural and hybrid tractability. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22–27, 2014*, pages 132–143, 2014.

[47] Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM (JACM), Volume 54, Issue 1, Article No. 1*, March 2007.

[48] Martin Grohe. *Descriptive Complexity, Canonisation, and Definable Graph Structure Theory*. Lecture Notes in Logic. Association for Symbolic Logic in conjunction with Cambridge University Press, to appear. Preliminary version available at `https://www.lii.rwth-aachen.de/de/13-mitarbeiter/professoren/39-book-descriptive-complexity.html`.

[49] Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. Deciding first-order properties of nowhere dense graphs. *J. ACM*, 64(3):17:1–17:32, 2017. Conference version: in Proceedings of the 46th ACM Symposium on Theory of Computing (STOC'14), pp.89–98, 2014.

[50] Martin Grohe and Nicole Schweikardt. First-order query evaluation with cardinality conditions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'18, Houston, TX, USA, June 10–15, 2018*, 2018.

[51] Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable? In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, STOC'01, Heraklion, Crete, Greece, July 6–8, 2001*, pages 657–666, 2001.

[52] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, pages 358–369, 1995.

[53] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD'93, Washington, D.C., USA, May 25–28, 1993*, pages 157–166. ACM, 1993.

[54] Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti A Zadeh. *Feature extraction: foundations and applications*, volume 207. Springer, 2008.

[55] Lucas Heimberg, Dietrich Kuske, and Nicole Schweikardt. Hanf normal form for first-order logic with unary counting quantifiers. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5–8, 2016*, pages 277–286, 2016.

[56] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, 2001.

[57] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing, STOC'15, Portland, OR, USA, June 14–17, 2015*, pages 21–30, 2015.

[58] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 137–152, 2015.

[59] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD'17, Chicago, IL, USA, May 14–19, 2017*, pages 1259–1274, 2017.

[60] Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science*, 7(2), 2011.

[61] Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA, June 22–27, 2013*, pages 297–308, 2013.

[62] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 325–340, 2018.

[63] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst-case and beyond. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS'15, Melbourne, Victoria, Australia, May 31 – June 4, 2015*, pages 213–228, 2015.

[64] Benny Kimelfeld and Christopher Ré. A relational framework for classifier engineering. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 5–20, 2017.

[65] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[66] Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles*

of Database Systems, PODS'10, June 6–11, 2010, Indianapolis, Indiana, USA, pages 87–98, 2010.

[67] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[68] Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS'16, San Francisco, CA, USA, June 26 – July 01, 2016*, pages 75–90, 2016.

[69] Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 223–234, 2011.

[70] Dietrich Kuske and Nicole Schweikardt. First-order logic with counting: At least, *weak* Hanf normal forms always exist and can be computed! In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'17*, 2017. Full version available at CoRR abs/1703.01122.

[71] Alon Y. Levy, Divesh Srivastava, and Thomas Kirk. Data model and query evaluation in global information systems. *J. Intell. Inf. Syst.*, 5(2):121–143, 1995.

[72] Leonid Libkin. *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[73] Leonid Libkin and Limsoon Wong. New techniques for studying set languages, bag languages and aggregate functions. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota, USA*, pages 155–166, 1994.

[74] Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 67:1–67:10, 2014.

[75] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982.

[76] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM (JACM), Volume 60, Issue 6, Article No. 42*, November 2013.

[77] Bernard M. E. Moret and Henry D. Shapiro. *Algorithms from P to NP: Volume 1: Design & Efficiency.* Benjamin-Cummings, 1991.

[78] Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22–27, 2014*, pages 234–245, 2014.

[79] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'12, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48, 2012.

[80] Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 179–191, 2018.

[81] Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD'16, San Francisco, CA, USA, June 26 – July 01, 2016*, pages 511–526, 2016.

[82] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.

[83] Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997.

[84] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael J. Cafarella, Michael I. Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 283–284, 2015.

[85] Sylvio Rüdian. So funktioniert die seo-ki, 2019. (In German) Available at `https://seorld.com/seo-ki`, Accessed: 2019-09-18.

[86] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 3–18, 2016.

[87] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO queries over nowhere dense graphs. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'18, Houston, TX, USA, June 10–15, 2018*, 2018.

[88] Thomas Schwentick and Thomas Zeume. Dynamic complexity: recent updates. *SIGLOG News*, 3(2):30–52, 2016.

[89] Detlef Seese. Linear time computable problems and first-order descriptions. *Mathematical Structures in Computer Science*, 6(6):505–526, 1996.

[90] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory, ICDT'13, Genoa, Italy, March 18–22, 2013*, pages 10–20, 2013.

[91] Luc Segoufin. A glimpse on constant delay enumeration (invited talk). In *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science, STACS'14, March 5–8, 2014, Lyon, France*, pages 13–27, 2014.

[92] Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.

[93] Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In *20th International Conference on Database Theory, ICDT 2017, March 21–24, 2017, Venice, Italy*, pages 20:1–20:16, 2017.

[94] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 318–329, 1996.

[95] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of the 17th International Conference on Database Theory, ICDT'14, Athens, Greece, March 24–28, 2014*, pages 96–106, 2014.

[96] Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. In *Proceedings of the 17th International Conference on Database Theory, ICDT'14, Athens, Greece, March 24–28, 2014*, pages 38–49, 2014.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Dr. Nicole Schweikardt for the continuous support of my Ph.D study, her patience, motivation and offering me a place in her group. Her guidance helped me in all the time of research and writing of this thesis.

I thank Prof. Dr. Stefan Kratsch and Prof. Dr. Wim Martens very much for serving as a reviewer of this thesis.

Of course, I want to thank to all my former and current colleages in the research group "Logic in Computer Science" for their support and the agreeable atmosphere. In particular, I want to thank Gesine Pergl and Eva Sandig for the administrative support and Petra Kämpfer for the technical support.

Moreover, I thank Philipp Badenhopp, Benedikt Konstantin Gräßle, Michael Rücker, Sylvio Rüdian, Benjamin Schlotter, Dr. Markus Schmid, Dr. Sandra Schulz and Viktor Well for reading parts of this thesis, their helpful comments and their support.

Sebstständigkeitserklärung

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad.

Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42 am 11. Juli 2018, habe ich zur Kenntnis genommen.

Berlin, den 12. Dezember 2019